The Report Committee for John Patrick Tourish
Certifies that this is the approved version of the following report:

# dspIP: A TCP/IP Implementation for a Digital Signal Processor

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Andreas Gerstlauer

Christine Julien

# dspIP: A TCP/IP Implementation for a Digital Signal Processor

by

John Patrick Tourish, B.A; B.S.E.E.

## Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

## Master of Science in Engineering

The University of Texas at Austin

May 2011

# Abstract

## dspIP: A TCP/IP Implementation for a Digital Signal Processor

John Patrick Tourish, M.S.E.

The University of Texas at Austin, 2011

Supervisor: Andreas Gerstlauer

From the initial implementations for the DEC PDP-11 to those of today done for commodity PICs, the TCP/IP code stack continues to work its way into a smaller and more omnipresent class of devices. One shortcoming of current devices on the leading edge of this trend is that they belong more to the microcontroller categories, which typically lack any appreciable signal processing capability. Applications such as consumer electronics and wireless sensor networks could benefit greatly from single-chip network-capable devices which are based on a Digital Signal Processing (DSP) core rather than a microcontroller. This report details the design and implementation of a partial TCP/IP code stack intended for such a DSP.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1: Introduction

Although TCP/IPv4 as detailed in RFC 791 and RFC 793 was not published until 1981 [1,2], initial implementations were already being developed in the late 1970's on PDP-11 class minicomputers [3]. More complete and sophisticated implementations on workstation-class machines followed in the late 1980's with the emergence of BSD 4.3 with Tahoe and Reno congestion control [4].

The first TCP/IP implementations for embedded devices appeared in the late 1990's [5-6]. More recent embedded devices from the late 2000's (e.g., ARM Cortex M3, Atmel AVR32, Microchip PIC32) all run lightweight TCP/IP code stacks such as lwIP, or the Microchip TCP/IP stack, and are small and inexpensive, but these devices also lack any appreciable signal processing power which limits their application space. Thirty years after the initial implementations, TCP/IP continues to be pushed into an ever smaller and omnipresent yet unseen, class of network-connected devices. The motivation for creating the solution described in this report is the author's desire to create a deeply embedded TCP/IP solution with high computing capability that does not require an external SDRAM and will have a bill of materials (BOM) of less than $5.

Although these smaller devices have greater processing power than the DEC minicomputers used for initial TCP/IP implementations (50+ MIPs vs. <1 MIPs) and comparable memory sizes (~128 Kbyte), they do not benefit from the extensive OS support such as sockets, virtual memory, and file systems that was a boon for the 1980's workstation-class implementations, and yet, the throughput requirement for many modern applications can be much higher, >1 Mb/s vs. <100 Kbit/s. Also, these embedded devices often require an application which needs to leverage and coexist with the network connectivity that is more computationally intensive than the email and ftp applications of

the past. In addition, the existence or lack of an efficient C compiler can be a major factor in the ability to port software to these embedded devices.

These technical constraints present challenges in developing or porting TCP/IP stacks for these embedded devices. Microcontrollers based on the ARM Cortex M3, the Atmel AVR32, and the MIPS M4k cores are probably the best examples of what is currently attainable from embedded microcontroller devices with clock speeds of 66-80 MHz, an integrated 10/100 Ethernet MAC/PHY, 256-512 Kbytes flash, 64-96 Kbytes of SRAM, and good C compilers to enable fast and efficient porting of existing code stacks. These devices ship with lwIP or the Microchip TCP/IP code stack. One other point which needs mentioning regarding porting of existing code stacks is the licensing agreement specified within the code. The Microchip TCP/IP stack is only available for use on Microchip devices. Any code stacks which are governed by a GPL license will not be able to be used legally within any non-GPL software. lwIP is governed by a BSD style license, which gives it a much broader appeal for incorporation into any proprietary code stack.

There certainly exists a class of applications which would benefit from the increased signal processing capabilities that a Digital Signal Processor (DSP) based solution would provide, relative to a microcontroller. These possible applications for a network-enabled DSP include: a single chip Internet radio, an embedded Skype phone, and sensor networks that rely on heavy signal processing at the sensor nodes.

Typically, DSPs have had a much smaller memory footprint than microcontrollers, or have been cache-based designs relying on external SDRAM. The smaller memory footprint devices would not be able to accommodate a TCP/IP stack, or if so, not concurrently with a network-enabled application. The cache-based designs with

external memory suffer from greater complexity and higher cost unfavorable for deeply embedded applications. Recently, DSPs implemented in smaller geometries have yielded devices with memory footprints comparable to the current lightweight microcontrollers and at similar costs, which should allow developers to implement a TCP/IP stack alongside a signal processing intensive network-enabled application.

Fixed-point DSP architectures will typically perform fractional arithmetic natively as is common in signal processing. This fact, combined with the lack of byte addressability and a non-byte size for the 'char' data type can greatly complicate the porting of code written in C. It also complicates the development of a TCP/IP code base written in the native DSP assembly, as manipulating TCP/IP header structures with such a machine architecture requires the use of logical shifts, ANDs, and ORs to accomplish what can easily be done on a microcontroller with 8-bit char size and byte addressability. On the other hand, the DSP architecture can be leveraged in performing the more computationally intensive tasks for TCP/IP, with the checksum calculation being the prime example. With the DSP used for this project, the main loop for calculating the one's complement checksum can be performed across 32-bit words in a single clock cycle due to support from the a zero-overhead hardware do-loop.

This report details the modeling, design, and implementation of a TCP/IP code stack for a proprietary DSP. Since there is no integrated Ethernet MAC/PHY on this device, this project will make use of a Microchip ENC28J60 Ethernet controller, and interfacing to the DSP will be done via a Serial Peripheral Interface (SPI) bus.

### 1.1 Current Solutions and Related Work

Partial TCP/IP implementations for deeply embedded processors have existed since 1999 [5-6], and complete implementations for DSPs since 2001 [7].

Currently available embedded TCP/IP solutions will now be examined in the light of our project goal that seeks to define a low-cost high-MIPs single chip solution. The specific devices we will compare are the following:

- Atmel AVR 32-bit AT32UC3A1512
- Texas Instruments (TI) 32-bit ARM Cortex M3-based Stellaris LM3S5B91
- 16-bit TI MSP430F6438
- 32-bit MIPS M4K based Microchip PIC32MX775F256H
- TI TMS320DM6413 32-bit DSP

Both the Atmel AVR series as well as TI MSP430 family are popular solutions for the sensor network application space. Table 1 lists some of the relevant features for each of these devices.

We need to qualify our definition for a "single chip" solution, mainly with regard to integrated flash. Integrated flash, while present on most of the devices under consideration, and desirable from a system size and complexity perspective, may actually be contrary to some of the other stated goals of this project, and thus will not be considered a necessary feature for the solutions we will be investigating. Specifically, code running from the integrated flash can limit the device clock speed at which single cycle instruction access is possible. For example, the Atmel AVR can achieve single cycle access up to 33 MHz, while the TI Stellaris can reach speeds up to 50 MHz). Also, it is possible that the integrated flash can have fewer available erase/write cycles than dedicated devices; 100,000 cycles are typical for standalone serial flash devices, whereas the flash in the TI Stellaris devices have an issue limiting them to 100 cycles [8]! Given that a 4 Mbit (512 Kbyte) serial flash device costs about $1, the author feels using an

external flash device is a good trade-off that will not sacrifice MIPs or program/data integrity.

There are a couple of other points to be made regarding features **not** included in the comparison. One point is that although a wireless solution may be preferable (or required as for Wireless Sensor Networks), and some of these solutions have the capability to interface with wireless adjunct devices, we are only considering wired Ethernet for this comparison. A follow-up project can focus on the comparison of wireless solutions. One other pertinent metric which would immediately follow the wireless capability is power consumption. Again, although this is an extremely important parameter for WSN and mobile applications, this parameter requires complex analysis, and is beyond the scope of this report and will be left for the follow-up project as well.

Other features, such as integrated A/D, DMA, timers, serial ports, development tool chain, and quality or capabilities of the TCP/IP code stack, while certainly important (or required) for specific applications, will also not be considered for the comparison.

Our comparison focuses on the signal processing capability of the devices. When comparing devices, the use of the term "MIP" as a measure of system performance, is often a vague metric that fails to account for differences in the architectures of each machine with regard to word width, pipelining, instruction cycle counts, and VLIW architectures which can achieve multiple-issue instructions to various functional units, such as the TI TMS320DM6413 device and the DSP device used for this project. One metric which has become common and attempts to normalize these architectural differences in the comparison of such devices is the MIPs/MHz performance figure. This figure is usually derived from a given architectures' performance in running a fixed point benchmark suite such as Dhrystone. As can be seen from table 1, this figure is 1.25 to

1.5 MIPs/MHz for most of the microcontroller architectures, but can be higher for the VLIW based devices. Again, this is an average figure from a benchmark which may not well reflect the true performance of the actual application that runs on the end product.

Some of the microcontroller-based devices now have hardware for single-cycle multiplication and DSP-like instructions such as Multiply-Accumulate and Saturation. These instructions are very useful because the Multiply-Accumulate instruction is a cornerstone for the filtering algorithms commonly used in signal processing. Also, word width can become critical to obtaining a stable and accurate signal processing filter response, and 16-bit arithmetic is typically insufficient for most real world signal processing of 16-bit data.

If we impose an imprecise and arbitrary benchmark regarding the minimum MIPs requirement for our TCP/IP plus real time signal processing application space of say 50 MIPs, then it is clear that the MSP430 does not meet that requirement (and because it is a 16-bit device as well). The other three microprocessor-based devices (AVR, Stellaris, and PIC32) can achieve the minimum performance figure as long as we do not require significant use of 32x32 bit multiplies with 64-bit accumulation, for example, the TI ARM Cortex M3 based device takes 4-7 cycles for this instruction. The one device that easily meets the MIPs requirement is the TI TMS320DM6413 device. However, a solution based on this device would most likely require an external SDRAM device as the memory resource requirements for the NDK code stack exceed the available on-board memory [9]. None of the solutions meet our $5 total BOM cost requirement, although the Microchip PIC32 device comes closest.

| Device | Speed (MHz) | SRAM (Kbyte) | Flash (Kbyte) | Ethernet MAC/PHY | MIPs/MHz | TCP/IP Stack | Cost $ |
|---|---|---|---|---|---|---|---|
| Atmel AVR AT32UC3 A1512 | 66 | 64 | 512 | Yes | 1.5 | lwIP | 16.00 |
| TI Stellaris LM3S5B91 | 80 | 96 | 256 | Yes | 1.25 | lwIP | 8.65 |
| Microchip PIC32MX 775F256H | 80 | 64 | 256 | Yes | 1.5 | Microchip TCP/IP | 5.42 |
| TI MSP430 F6438 | 25 | 18 | 256 | No | ?? | uIP | 8.40 |
| TI TMS320 DM6431 | 300 | 32 L1P 64 L1D 64 L2 | NA | Yes | Up to 8 | TI NDK | 11.25 |

Table 1: MIPs and memory resources available with current embedded solutions.

There are basically four TCP/IP stacks which are used across these devices; lwIP, uIP, the Microchip TCP/IP stack, and the TI NDK code stack.

Texas Instruments has made their Network Developers Kit (NDK) available for their TMS320C6000 family of DSPs since 2001. This TCP/IP stack is rather large, as it

requires 200-250 Kbytes of program memory and 95 Kbytes of data memory [9]. Solutions using this code stack will require the use of external SDRAM, which is contrary to our goal for a single-chip solution. The TI NDK will only be available to run on TI TMS3206000-based DSPs. Likewise, the Microchip TCP/IP stack device will only be available to run on Microchip devices.

Probably the most important and pervasive implementations of TCP/IP for the embedded community are a pair of open source multi-platform solutions, lwIP and uIP, both of which have been in development by Adam Dunkels *et al* since 2001 [10, 11]. The uIP code stack was originally designed to address the deeply embedded 8-bit market with smaller memory resource requirements, and can be used in conjunction with the Contiki OS for solutions targeting Wireless Sensor Network (WSN) applications. The lwIP code stack is often used on devices with larger memory footprints. Both of these code stacks are governed by a BSD style software license, so they are attractive for integrators of proprietary software. Given the issues with porting a large C library to a DSP which has an inefficient C compiler and the DSP OS requirements that would be needed to interface with the lwIP API, the author decided not to use either uIP or lwIP for this project.

## 1.2 Project Goals

The principal goal of this project and report is to develop and demonstrate a network connectivity code stack on an existing DSP device coupled with an off-the-shelf external Ethernet controller connected via a SPI bus (see Figure 1) providing the physical layer.

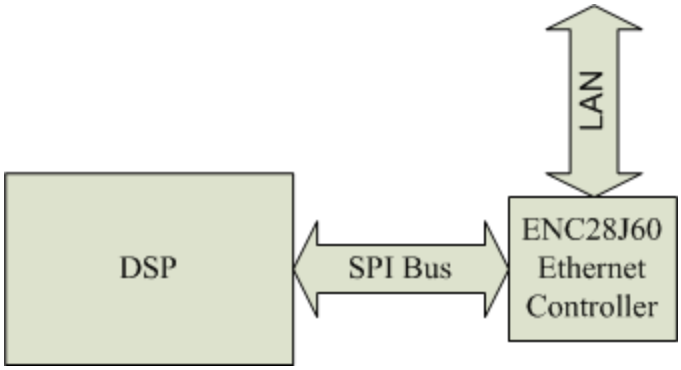


Figure 1: Block diagram showing DSP and Ethernet controller connected via SPI Bus.

Follow on DSP devices could be developed which integrate the Ethernet MAC/PHY for a lower cost solution, but the important milestone of demonstrating network connectivity with native DSP code regardless of where the physical layer lies is a proof of concept to show the capabilities and applications that such a device could support. To meet that end, the main tasks are to accomplish the following:

- Develop a hardware driver to enable the DSP to use the ENC28J60 as its physical layer interface. This step entails writing DSP code to configure the various registers of the ENC28J60 as well as sending Ethernet packets to and receiving them from the 8 Kbyte FIFO contained within this Ethernet controller. The principal sources used to develop the driver were the Microchip ENC28J60 Data Sheet [26] and Errata [27] documents, as well as the driver code contained within the Microchip TCP/IP code stack. Note that no other Microchip source code from the TCP/IP code stack itself was used for this project as per the license agreement (clause (ii) was applicable to the usage for this project):

  Microchip licenses to you the right to use, modify, copy, and distribute:

(i) the Software when embedded on a Microchip microcontroller or digital signal controller product ("Device") which is integrated into Licensee's product; or

(ii) ONLY the Software driver source files ENC28J60.c, ENC28J60.h, ENCX24J600.c and ENCX24J600.h ported to a non-Microchip device used in conjunction with a Microchip ethernet controller for the sole purpose of interfacing with the ethernet controller.

- Develop a TCP/IP network connectivity code stack. This will be developed from the ground up in the native DSP assembly language, using Internet Engineering Task Force (IETF) Request for Comment (RFC) documents as the principal source for developing code. Other sources consulted in the design and development included Wireshark packet captures of the various protocols using existing implementations (e.g. Windows XP), as well as the lwIP and uIP open source TCP/IP code stacks. Given the time and resource constraints for this project, this project will not be a full implementation as specified by RFC 1122 – "Requirements for Internet Hosts", but will implement enough of the protocol stack to be able to function on a local network. One of the more memory intensive yet required (as per RFC 1122) components which will not be implemented is that of re-assembly and fragmentation of IP datagrams.

- Integrate the TCP/IP code stack with the existing DSP Operating System (OS) and add the capability for interfacing TCP/IP with DSP applications. This OS is a proprietary simple data-streaming OS which does not have many features normally found in a full-featured RTOS.

- Demonstrate the network connectivity by echoing packets delivered via UDP and TCP (echo server).

## 1.3 Project Methodology

The first choice in methodology was to decide whether to port an existing TCP/IP code stack such as lwIP or uIP, or to develop one from the ground up. The factors influencing that decision included the portability of the lwIP code base to the existing OS architecture, i.e. how well the lwIP API matches up with the existing OS features (e.g. IPC and semaphores), as well as the capability of the C compiler for the DSP device itself (as lwIP is written in the C language). The C compiler for this device was recently developed, and so was not seen as mature enough for developing a component robust enough to be integrated with the OS. The use of a C compiler would also entail some code bloat above a native assembly language implementation, which again is anathema for memory-restricted devices.

Even given a mature and efficient C compiler, the choice of integrating a general purpose multi-platform code stack such as lwIP brings with it extra features which may not be needed, and would lack optimizations for the given DSP architecture used in this project. These extra features can be removed and optimizations performed, but at that point, the time-to-market benefit of choosing the re-use route begins to vanish.

The final porting decision was whether "hand porting" of individual routines and protocols from lwIP made sense, i.e. having the programmer manually translate C language routines to DSP assembly. This type of hand porting can be effective for some C routines, and was done for a small portion of the ENC28J60 driver code where register initializations are easily mapped from one language to the other, but this style of

programming may not do as well for large scale state machines and does not help the API OS interface issue mentioned above.

Given all of these factors, and given that the RFC documentation from the IETF, while terse for some of the earlier protocols, is actually sufficient for this implementation, the author decided to develop the code stack from first principles as specified in the IETF RFC documents, supplemented with real world observations of packet traffic of existing systems.

To gain confidence with the Ethernet controller, and also to demonstrate concrete progress, the author decided to develop the protocol stack from the lowest layer and work upwards, with some form of unit testing done as each layer was brought up. This layered development approach dovetails nicely with the layered architecture of the protocol stack itself. The following steps were performed in this bring-up phase of the project:

- First, developed the device driver to enable the data link layer whereby writing and then self checking b y reading back these initialized control registers and onboard FIFO of the Ethernet controller was performed. Besides the TCP protocol itself, this layer took the longest amount of time to develop and debug.
- Once this unit test for the ENC28J60 was complete, the next step was to transmit an actual packet, and the simplest one to construct is the ARP (Address Resolution Protocol) request.
- Once the ARP request packet could be observed on the network using Wireshark, the next step was to add the IP layer so that an ICMP Echo packet could be transmitted to a local host.

- Once this packet was successfully transmitted, the receiving host issued an ARP request to get the MAC address of the initiator of the ping message (the DSP), and it was at this point that the packet receive logic was implemented so that complete send/receive sequence could be performed.
- Once the ARP request from other hosts on the LAN was handled correctly, the UDP protocol was developed so that an UDP Echo server could be implemented on the DSP in order that a continuous stream of packets be received and transmitted. This UDP Echo server milestone marked what was considered the end of the bring-up phase of the project.

## 1.4 PROJECT TOOLS

Various tools were used throughout the project, and the essential development board, assembler, linker, and GUI-based debugger were included as part of the Evaluation Kit provided by the DSP manufacturer. The debugger has standard features found in graphical debuggers, such as setting breakpoints, single stepping, viewing and modification of the DSP registers and memory. A debugger with these higher level features allows for more bugs to be discovered and addressed in a single assemble/link/download debug session, since one can uncover one bug, but temporarily work around it by modifying register or memory contents with a correct result, and then continue stepping through code to the next issue, etc.

The next most important tool for the bring-up of the driver for the Ethernet controller was the inexpensive USB-based logic analyzer from Saleae Logic. As explained in the Chapter 4, this tool allowed for easy visualization of the SPI traffic so

that errors in the code and timing could be identified and fixed more quickly than if this tool had not been used.

Once a packet was ready to transmit onto the network from the development platform, the next important tool, Wireshark, was used throughout the rest of the project to analyze network packets. Wireshark is able to analyze all of the bytes of an Ethernet packet, and since it has inherent knowledge of all of the TCP/IP protocols and header fields, it can quickly show the contents of each layer in the protocol. A tool called EchoTool Echo Server was used as the packet generating companion to Wireshark to produce packet traffic for debugging, unit testing and demonstration purposes.

Because the ENC28J60 does not auto-negotiate full duplex (i.e. both sides need to be manually setup for full duplex), a managed switch (Cisco SG300-10) was used so that configuration of the full duplex setting on that end of the network could be performed. Much of the errata for the ENC28J60 are associated with packet collisions which occur in half-duplex mode [27], so ensuring full duplex helped to avoid the coding up the software workarounds that are needed for proper device behavior.

# Chapter 2: DSP Device Overview

## 2.1 DSP Architecture

The DSP used for this project is a cacheless Modified Harvard Architecture 32-bit fixed point device with three address spaces, using one for program instructions, one for X-data, and one for Y-data. This architecture is commonly used in DSPs, as the separate X and Y memory spaces can be used to hold input data and filter coefficients that can then be accessed in parallel to feed a Multiplier or Multiply-Accumulate Unit. Figure 2 contains a block diagram of the top-level DSP architecture.

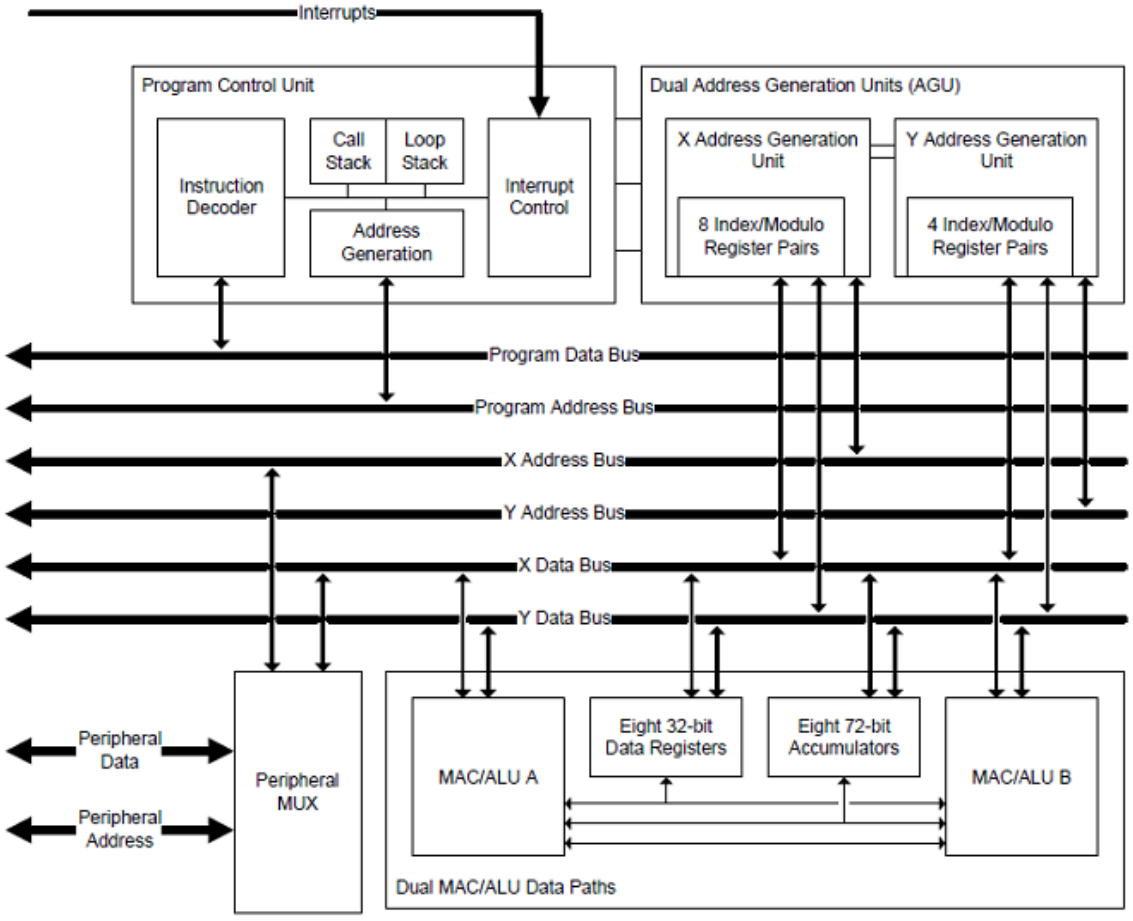


Figure 2: DSP Architecture showing independent P/X/Y address spaces.

The DSP contains eight 32-bit data registers for holding input data and coefficients, and eight 72-bit registers to hold the results of Multiply, Multiply-Accumulate (MAC), and Arithmetic-Logic Unit (ALU) instructions. There are also twelve 16-bit registers used for addressing the 16-bit address space of the DSP. Figure 3 shows more detail of the data paths between the 32-bit data registers, the MAC and ALU, and the 72-bit accumulator registers.

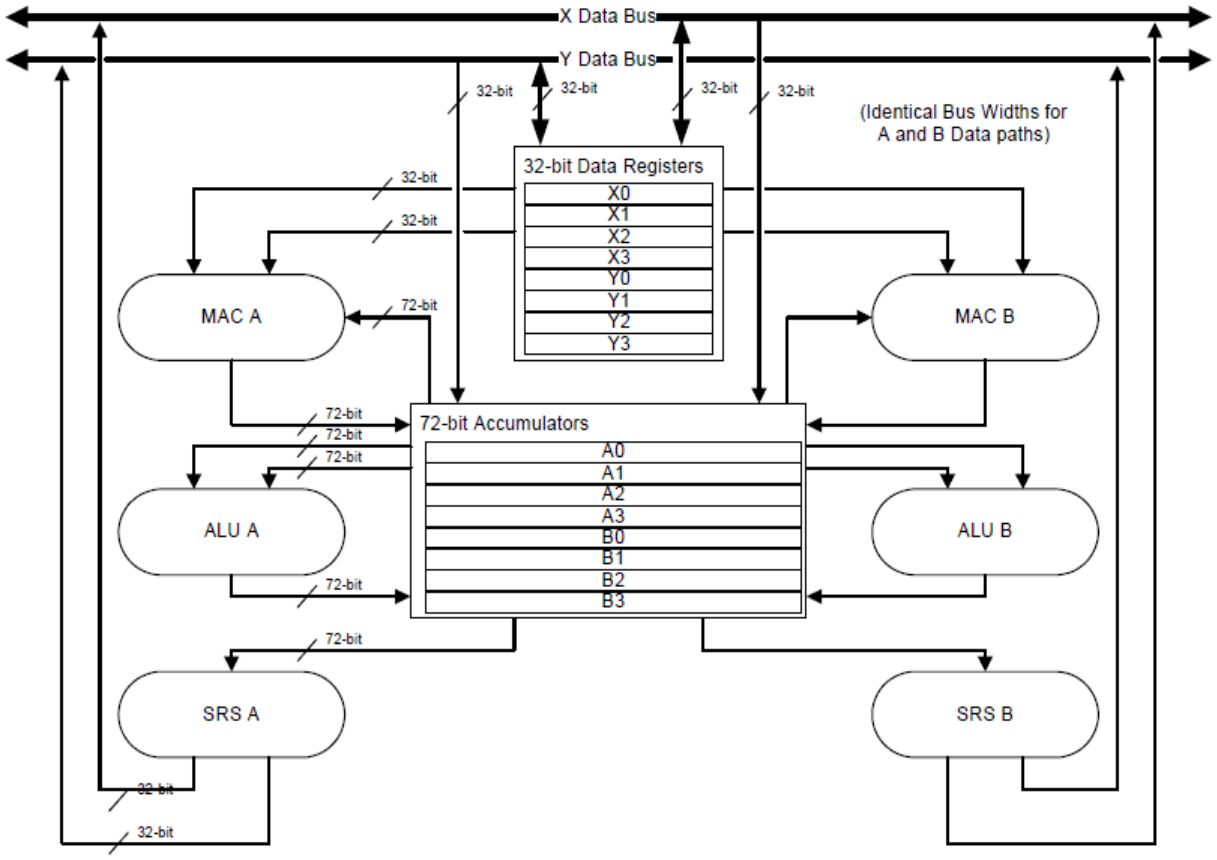


Figure 3: DSP data registers and accumulators, MAC and ALU units.

As is typical for fixed point DSP architectures, the native arithmetic of the machine is fractional, with numbers being represented in the range from -1.0 to $1.0 - 2^{-31}$,

and denoted as 0x80000000 to 0x7fffffff in 32-bit hexadecimal format. One other point regarding the number format is that the DSP uses a big-endian ordering. Since the network byte order as specified by the Internet Protocol is big-endian, this eliminates the complexity and wasted MIPs for the conversion required on little-endian machines.

The amount of memory integrated on the device varies with the target cost and application market, ranging anywhere from 24 Kwords to 180 Kwords, spread across the P/X/Y memory spaces. Likewise the device speed can vary from 80 MHz to 300 MHz, again depending upon the target application space.

The main compute engine is comprised of the dual MAC/ALU units, but there are also Address Generation Units (AGU) and Load/Store Units as well. These functional units can operate in parallel, and can be driven with a single 32-bit instruction word dispatch. Up to six MIPs/MHz can be achieved by using these functional units in a parallel fashion. Figure 4 shows an example of such a single VLIW (Very Long Instruction Word) instruction, with dual MAC, dual AGU, and dual memory moves occurring in a single clock cycle (notice the C language instruction style syntax).

```
a0+=x0*y1; b0+=x0*y0; y0=ymem[i4]; i4-=1; x0=xmem[i0]; i0+=1
```

Figure 4: Example DSP instruction showing six parallel operations.

Even though the DSP architecture is optimized for signal processing tasks, there are instructions available for use which are “RISC-like” in nature and include instructions for bit level manipulation, such as the setting and clearing of bitfields within 32-bit data words, as well as instructions for logical operations such as shifting, ANDing, ORing and XORing. These instructions ease the implementation of microcontroller types of tasks

such as handling host communication, and for the purpose of this project, the coding of the TCP/IP stack. The use of some of these instructions can limit the combination of the parallel DSP units, and so the 6 MIPs/MHz performance figure cannot be achieved across all instruction sequences. Figure 5 shows a short assembly langua ge routine using these RISC-like instructions to increment an ICMP header sequence number by one, and insert this 16-bit value into a 32-bit header field. This code also illustrates the issue mentioned in Chapter 1 regarding the non-byte, or in this case non-16-bit word, native data types and addressability, and the extra logical instructions needed to make up for this shortcoming.

```
X_S_ICMP_Tx_Header_Sequence_Number:
        # pre-increment the sequence number
        b0 = xmem[X_VX_ICMP_Sequence_Number]
        uhalfword(b1) = (1)
        b0 = b0 + b1
        anyreg(x0,b0 h)
        bitclr hi(x0), (0xffff)   # sequence number is 16 bits
        xmem[X_VX_ICMP_Sequence_Number] = x0
        anyreg(b1,x0)

        b0 = ymem[X_VY_ICMP_Tx_Echo_Packet_Sequence_Number]
        b2 = xmem[X_VX_ICMP_Echo_Packet_Sequence_Clear_Mask]
        b0 = b0 & b2                    # zero out sequence number field
        b0 = b0 | b1
        ymem[X_VY_ICMP_Tx_Echo_Packet_Sequence_Number] = b0h
        ret
```

Figure 5: DSP instruction sequence showing RISC-like programming style.

Other features of the DSP include peripheral interfaces such as Digital Input and Digital Output $I^2S$ (Integrated Interchip Sound) interfaces, Direct Memory Access (DMA) channels, hardware timers, Serial Control Ports (SCP), and a Programmable

Interrupt Controller (PIC). All instructions, with the exception of conditional branching and the accessing of peripheral registers, which control the configuration and reading/writing of the peripheral interfaces, have single cycle throughput. The SCP is commonly used for communication with a master host microcontroller or with a slave serial flash device using the SPI protocol. For this project, we will make use of one of the SCP interfaces to communicate with the slave Ethernet controller.

The DSP core also contains a nestable hardware do-loop, which enables loops to be performed without any overhead for the loop counter management and branching. A hardware-based call stack enables standard modular subroutine programming.

## 2.2 Operating System Overview

The OS which normally runs on the DSP device is a data-driven design optimized for the decoding of compressed audio streams such as Dolby AC3 or MP3, along with the post processing of the decoded audio stream using modules for mixing, filtering, delay, and volume control. Audio streams, whether a compressed bitstream or uncompressed PCM is brought into the chip via an input $I^2S$ interface, which is a 3-line interface comprised of a bit clock, sample clock, and data signals, and using a DMA channel to move the sample stream into data memory. Likewise, a DMA channel is used to transmit the final output stream via the output I2S interface to a DAC (Digital Audio Converter).

The main functions of the OS include the following:

- Setup and manage the input and output DMA channels for moving data
- Maintain buffer pointers for the movement of data across the DSP
- Call module entry points to operate on the data

- Handle host communication via a serial control port for the configuration of modules as well as for providing state information back to the host
- Handle interrupts for host communication, timers, etc.

The OS may contain multiple threads of execution, but this threading model does not resemble those used on modern operating systems such as Linux or BSD. Because it is necessary to keep buffering requirements, and latency, low in designs typical for this DSP, the uninterrupted flow of audio data translates into a hard real-time system, and so the threads of execution are delineated between the higher priority audio rate foreground processing thread, with everything else relegated to the background thread. The serial control port driver for interfacing with the Ethernet controller that is described in detail in Chapter 4 will execute in this background thread. Likewise, the TCP/IP stack itself will execute in the background thread as well.

Modules operating on the audio data run in a defined order and run to completion, and since the OS handles all of the access to the peripherals, there are only a couple of cases when resource sharing occurs, and those instances are easily handled with short critical sections. Hence, some full-fledged RTOS features such as semaphores and message passing are not necessary in the OS. Also, the memory allocation scheme used on the DSP does not have dynamic runtime capabilities but is done only upon startup. As discussed in Chapter 1, these optimizations can affect the ability to interface easily with generic libraries such as lwIP, which may have standard OS feature expectations.

# Chapter 3: TCP/IP Historical Overview

## 3.1 From NCP to TCP

An overview of the evolution of the TCP/IP protocol will now be presented as understanding this context is important in understanding key design decisions that went into the development of these protocols.

The predecessor to the modern Internet was the ARPANET, the first large scale packet switched network designed by researchers coming from MIT, implemented by Bolt Beranek and Newman, Inc. (BBN), and with funding provided by the Advanced Research Projects Agency (ARPA). The branch of ARPA responsible for this development is the Information Processing Techniques Office (IPTO), and the first director to be appointed to the IPTO in 1962 was J.C.R. Licklider from MIT. Licklider was the visionary who published earlier reports describing an "Intergalactic Computer Network", and these ideas were a motivating force that contributed to development of the ARPANET. Licklider persuaded other researchers to work for ARPA in the pursuit of building such a network. By 1968, a specification of a design for the ARPANET under the direction of chief scientist Lawrence Roberts (also from MIT) was completed. A Request for Quotations (RFQ) to build the nodes for the initial network was issued by ARPA in July of 1968 [12].

One of the main architectural differences between the modern Internet and the ARPANET as specified by Roberts is that the ARPANET did not support internetworking of heterogeneous networks, but rather, was composed of a single subnetwork of identical nodes called Interface Message Processors (IMP), to which heterogeneous host machines were attached (see Figure 6). Lease lines providing 50 Kb/s data rates connected the IMPs.

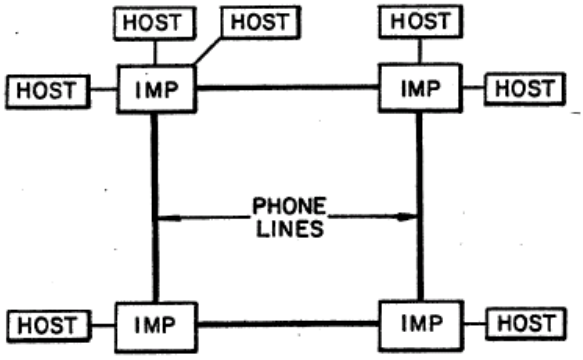


Figure 6: IMPs connected to form the ARPA network [13].

The ARPA contract to build the network was awarded to BBN, who used the Honeywell DDP-516, a 16-bit minicomputer with up to 32Kbytes of memory as the basis for the IMP [14]. By December of 1969, the first four nodes of the ARPANET were connected, consisting of an SDS Sigma 7 at UCLA, an SDS 940 at the Stanford Research Institute (SRI), an IBM 360 at UCSB, and a DEC PDP 10 at the University of Utah [13]. Soon after, IMPs continued to be added to the network at the rate of about one per month. Figure 7 shows a map of the ARPANET from April of 1971 detailing IMP locations with attached host computers.

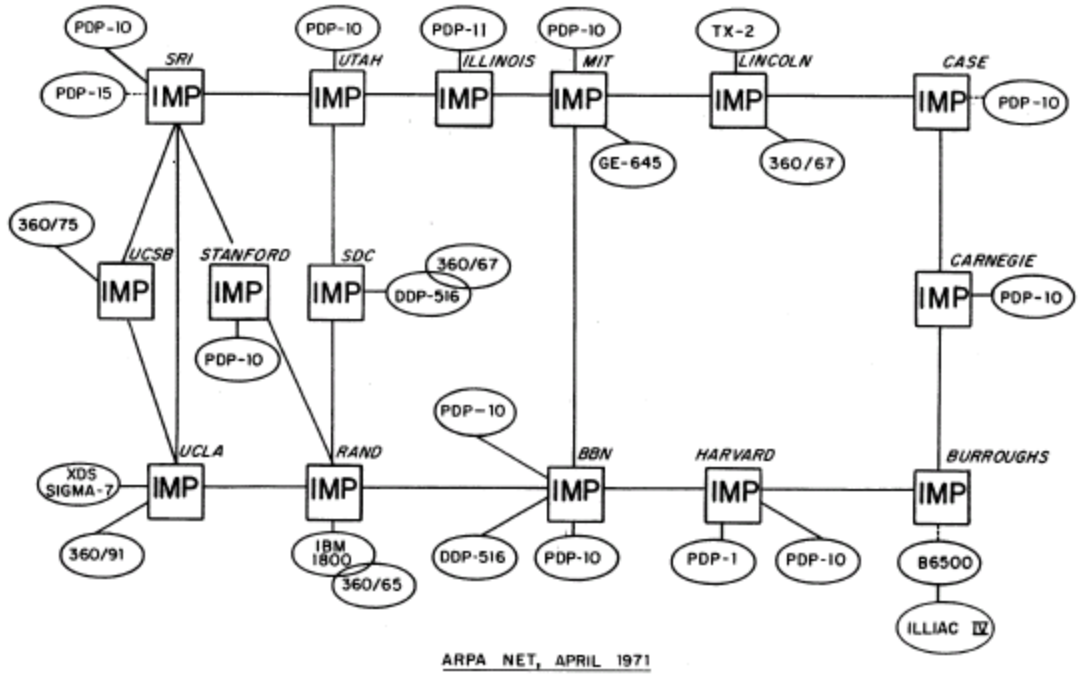


Figure 7: Early ARPANET map [13].

The ARPA RFQ details the list of responsibilities for the IMP [12]:

(1) Breaking messages into packets
(2) Management of message buffers
(3) Routing o f messages
(4) Generation, analysis, and alteration of formatted messages
(5) Coordination of activities with other IMPS.
(6) Coordination of activities with its HOST
(7) Measurement of network parameters and functions
(8) Detection and disposition of faults

To accomplish these responsibilities the RFQ also specified the initial Host-to-IMP and IMP-to-IMP protocols, and the actual BBN implementation was more thoroughly detailed in BBN1822 [14]. When a host wanted to send a message to another host, it sent the message to its IMP using the Host-to-IMP protocol. Messages could be up to 8095 bits in length, and were classified either as "Normal" messages, those intended for another host, or "Abnormal" messages, those intended as IMP commands. The IMP

would then take the message and divide it into packets of up to 1008 bits in length which would be routed to a destination IMP to which the destination host was attached. Hence the IMP was responsible for fragmentation and re-assembly of the messages from/to the hosts. The message packets would be sent from source IMP to destination IMP using the IMP-to-IMP protocol using Normal packets. Abnormal packets between IMPs were used for error control, acknowledgements, routing table information, etc. The normal packets containing the message contents were dynamically routed based on delay times for a given route and available buffering at a destination IMP [12].

In addition to the software running on the IMP for the Host-to-IMP and IMP-to-IMP protocols, ARPA also funded development of the Host-to-Host communications software as well, and the Network Working Group, a loose organization of researchers including UCLA graduate student Stephen Crocker, was formed in 1969 [15]. The NWG was effectively the predecessor to Internet Engineering Task Force (IETF). Crocker is also attributed with initiating the Request for Comments (RFC) memorandums that were instrumental in disseminating ideas in the early development of new protocols, and are still part of the standardization process for Internet protocols. By the end of 1970, the initial version of this Host-to-Host software, called the Network Control Program (NCP), was completed [15, 16]. NCP would become part of the operating system running on the host machines, and provide the transport layer on top of which applications such as FTP could be developed. Figure 8 shows the layering of the ARPANET protocol stack. Notice that the IMP-to-IMP protocol is not included in this stack diagram, as this was seen as being transparent from the Host's point of view.

The Host-to-Host protocol only allowed for one message at a time to be outstanding in the channel. When a message was delivered to the destination IMP, a

Ready for Next Message (RFNM) acknowledgment would be sent back to the source host. This one-message-in-transit approach curtailed the effective use of the network bandwidth.

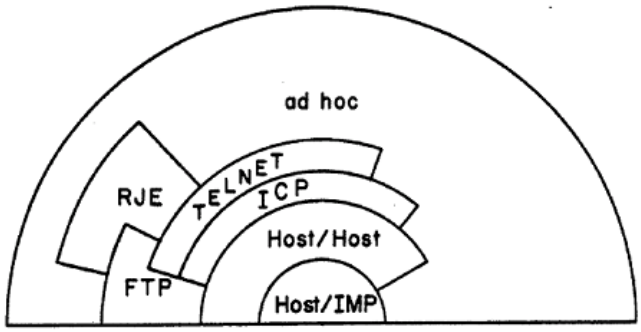


Figure 8: ARPANET protocol layering [13].

Unlike the gateways or routers of the modern Internet, the IMPs were responsible for much more message processing due to the fragmentation, re-assembly and reordering of messages, and this complicated the buffer management and exposed some flaws in the initial design. One such flaw was found early on when Robert Kahn, using a message generator to generate traffic, confirmed a deadlock condition whereby the IMP did not possess enough free memory to perform the re-assembly of a message. This design flaw was overcome by requiring the source IMP to allocate memory on the destination IMP before transmitting a fragmented message.

These problems and shortcomings in the ARPANET protocols was detailed as early as 1974 as in BBN Report No. 2918 [17]. Vinton Cerf also published in 1974 RFC 635 “An Assessment of ARPANET Protocols” [18], which maps out the motivation behind TCP:

> This paper deals with the motivations for the redesign of the HOST-to-HOST, IMP-to-IMP, and HOST/IMP communication protocols in the ARPANET.

Cerf, who together with Robert Kahn published the pivotal paper "A Protocol for Packet Network Intercommunication" [19], which provided the technical underpinnings for the TCP protocol that would succeed the protocols used in the ARPANET. Several versions of TCP were detailed in Internet Experiment Notes (IEN) with modifications of the initial design in the Cerf and Kahn paper, and in the final version, v4, the decision was made to split out a distinct internetworking layer, the IP layer [15].

Some of the design changes moving from the ARPANET protocols to TCP/IP included:

- Remove the message processing out of the subnetwork, this would become the responsibility of the software running on the hosts.
- Move communication reliability (packet acknowledgement, reordering, and retransmissions) out of the subnetwork and into the transport layer on the host. The internetworking layer would only provide a best effort approach to the delivery of packets.
- Provide a method of flow control between hosts
- Networks would be connected using gateways, or routers, and these would be stateless devices using addresses in the IP header to determine routing.
- Enable multiple packets to be outstanding in the network to maximize bandwidth utilization.
- Provide a global addressing capability
- Enable full duplex communication between hosts

Implementations for TCP/IP began appearing in the late 1970's [3], and the landmark RFC 791 for IP and RFC793 for TCP were published in 1981. A transition plan to move from NCP to TCP/IP was published in 1981 as well, with the complete

switch-over occurring on what is referred to as “Flag Day”, January 1, 1983 [18].

## 3.2 TCP/IP Protocol Layer Model

Similar to the early layering models of the ARPANET protocols, TCP/IP as detailed in RFC 1122 (Requirements for Internet Hosts -- Communication Layers), uses a four layer protocol stack with Application, Transport, Internet, and Link layers [21]. A protocol layering model which in some respects competed with the TCP/IP model was standardized by the International Standards Organization's (ISO), called the Reference Model for Open System Interconnection (OSI), and was originally published in 1984 [22]. The OSI model was criticized by some who viewed it as overly complex [23]. This model specified seven layers, and it is this protocol stack that is often the one that is cited when one speaks of the layering of protocols for communication networks, even though in reality, the TCP/IP model became the basis for implementations. The individual layers within TCP/IP and OSI protocol stacks are often compared and contrasted, and it is common for a mapping from one to the other to be presented even though some of the concepts between the two models differ. Figure 9 shows such a mapping between TCP/IP protocol layers and the OSI model [24]. Subsequent chapters of this report will detail portions of the lower three layers of the TCP/IP protocol stack.

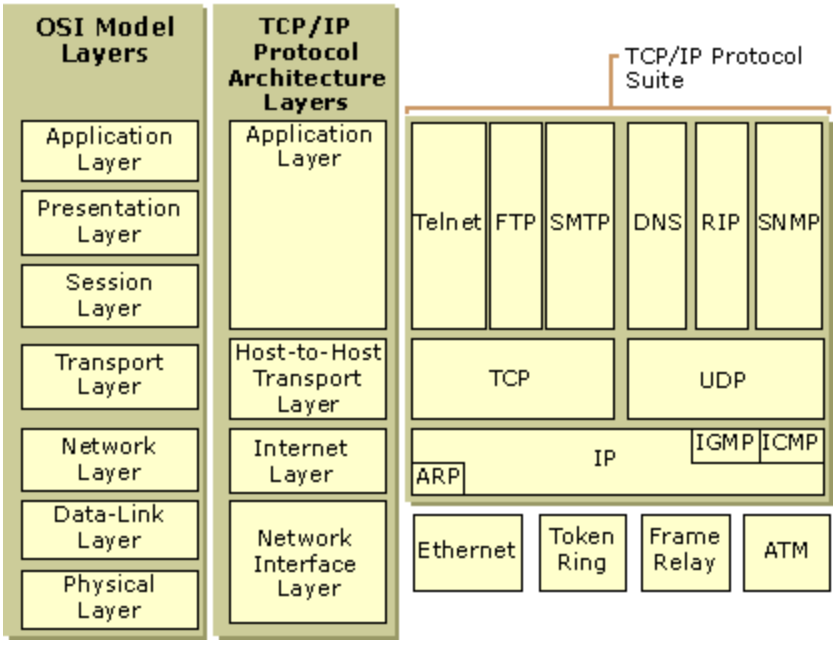


Figure 9: Side-by-side depiction of TCP/IP and OSI protocol layers [24].

TCP/IP protocol layering is implemented by successive encapsulation of application data by TCP segments, then by IP datagrams, and finally by the link layer Ethernet frame. Figure 10 graphically depicts this encapsulation.

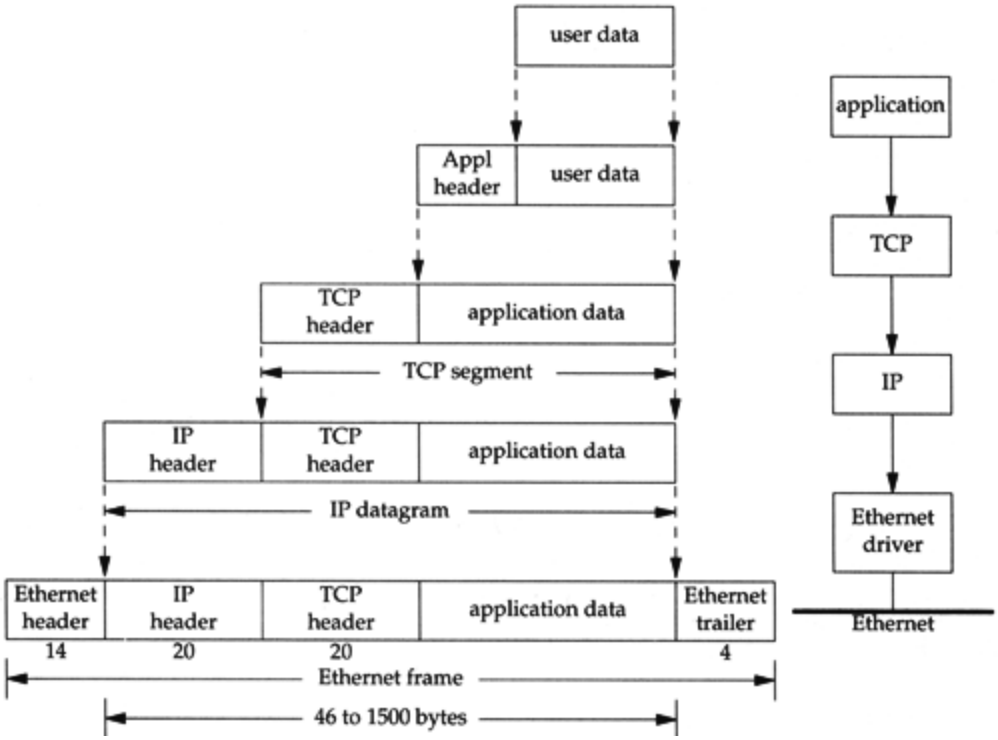


Figure 10: Encapsulation of TCP segment, IP datagram within the Ethernet Frame [4].

# Chapter 4: Link Layer

## 4.1 Ethernet Background

Ethernet is by far the most widely used link layer for wired network connectivity, and so was the logical choice as a solution for this project. Link layer technologies such as token ring (IEEE 802.5) and FDDI (Fiber Distributed Data Interface) were competitors to Ethernet back in the 1980's and 1990's, but have since become obsolete by Ethernet's advance into Gigabit speeds.

Invented in 1973 by Robert Metcalfe (who went on to co-found 3COM in 1979 and is now at the University of Texas) and David Boggs while at Xerox PARC, Ethernet is strongly patterned after the ALOHA protocol used in the first wireless data network, ALOHAnet. Originally, Ethernet was based on a shared medium, which used a single coaxial cable connecting multiple hosts in a bus topology for 10Base5 and 10Base2, where each node had equal access to the shared medium and transmitted in a half duplex mode. Figure 11 shows an example of this network bus topology. The first Ethernet specification was collaboratively published by Digital Equipment, Intel, and Xerox in 1980 and is referred to as the DIX standard [28]. This specification was submitted to the IEEE as the basis of the 802.3 standard which became finalized in 1985. Note that these specifications cover both the physical layer and the data link layer. There are many variants of the physical layer specified by various 802.3 sub-parts, using media such as coaxial cable, twisted pair cables, and fiber optic cables, and with speeds ranging from 10 Mb/s to 100 Gb/s. Most of the following discussion is specific to the original 10 Mb version.

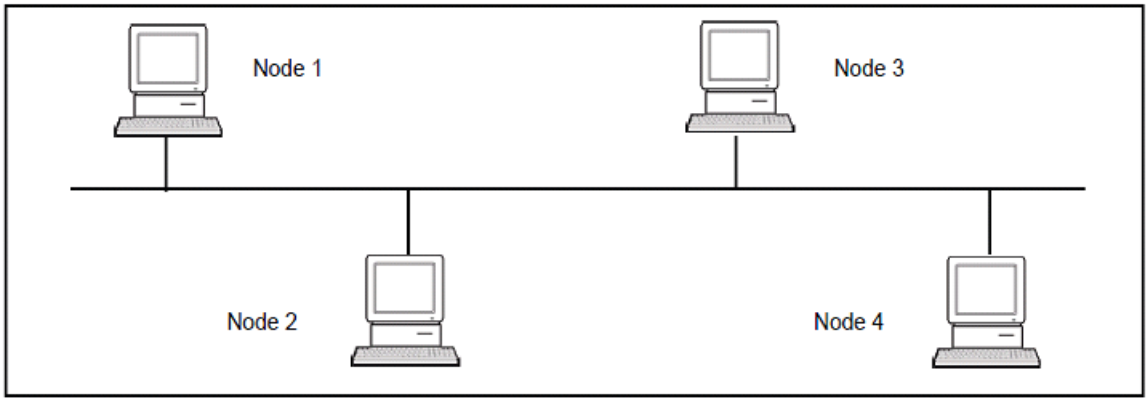


Figure 11: Ethernet network with bus topology [25].

The basis of the Ethernet protocol that enables multiple hosts to share the medium for a bus network topology is called Carrier Sense Multiple Access with Collision Detect (CSMA/CD). The Carrier Sense portion of the protocol means that each node is capable of detecting whether another node is currently transmitting. Before transmitting a frame, a node must wait until the medium is in an idle state, and then wait an additional 9.6 μsec period called the Inter-Packet Gap. The Multiple Access portion of the protocol means that each node has the capability to transmit on the shared medium. The Collision Detect portion of the protocol means that a node can detect if it and another node are transmitting at the same time. Collision detection requires that a node monitor the medium as it is transmitting, which is a matter of monitoring its Rx line. If a node detects a collision while transmitting, it will send a special jam signal to alert all other nodes. Once this occurs, exponential back-off will be used by each node before attempting to re-transmit.

More common network topologies for current Ethernet usage are to have a star network topology using a switch or router connecting the nodes. Figure 12 shows an example of a star network topology.

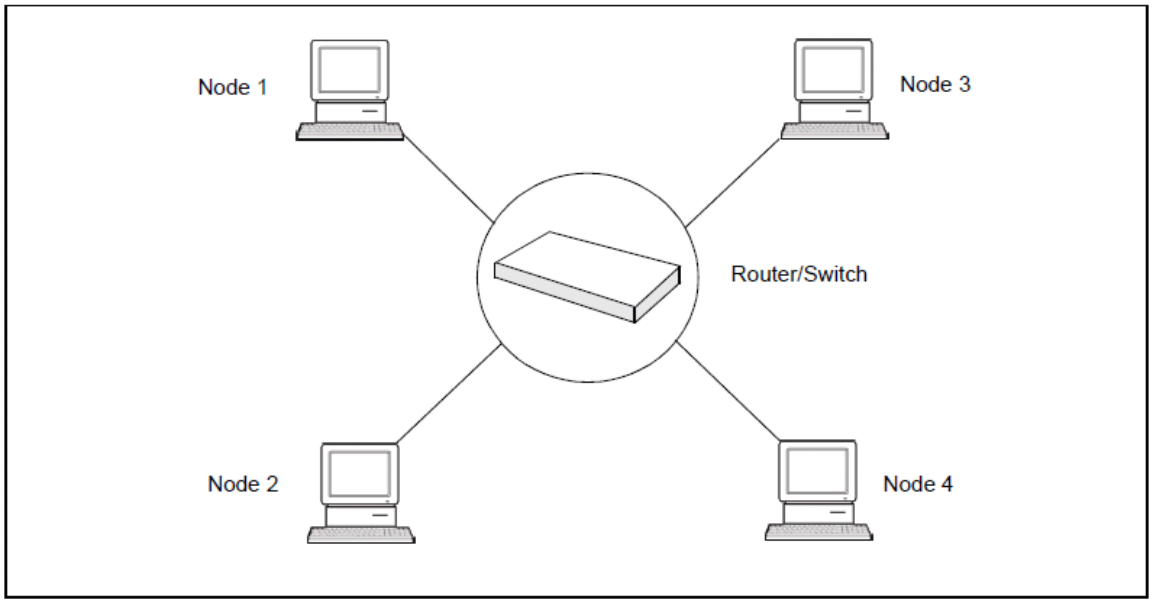


Figure 12: Ethernet network with star topology [25].

Such a star network topology eliminates the need to use CSMA/CD, and hosts can operate in a full duplex mode, doubling the bandwidth.

Figure 13 shows the details of an Ethernet frame. Each frame begins with a 7 octet preamble with repeating 0x55 that allows synchronization. The Destination and Source addresses are also known as the MAC (Medium Access Control) Address, or hardware address. These addresses are 48-bits in length and divided into two 3-octet fields, one called the Organizationally Unique Identifier (OUI), with the other used as the Hardware Identifier. The Type field contains the EtherType key use for demux, and for our purposes will be either 0x0800 for IP packets, or 0x806 for ARP packets. The Data payload follows the type field, and can be up to 1500 bytes. The minimum payload is 46 bytes, which corresponds to a 64 byte minimum frame size. Ethernet controllers will typically add padding to ensure minimum frame sizes.

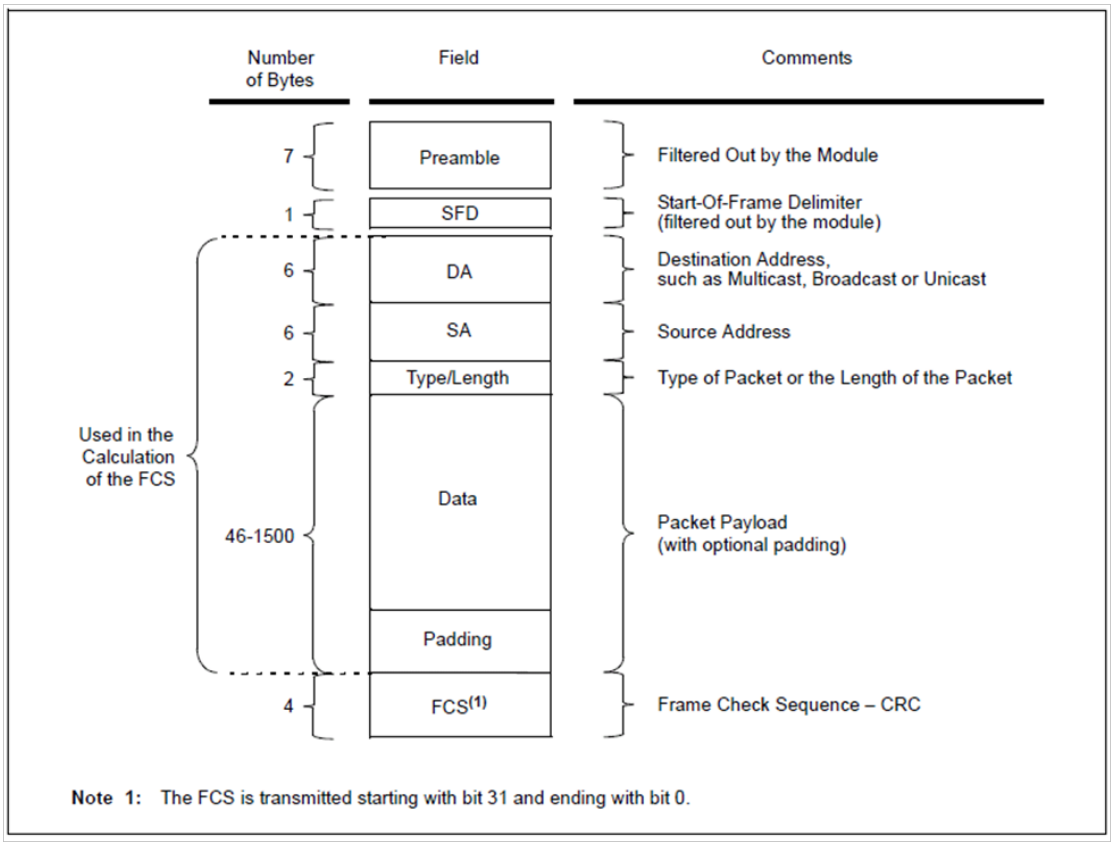


Figure 13: Ethernet frame structure [26].

## 4.2 ENC28J60 ETHERNET CONTROLLER

Since the DSP device used for this project does not have an integrated Ethernet MAC/PHY, an off-the-shelf Ethernet controller device from Microchip Technology, Inc., the ENC28J60, offers what is probably the lowest cost bolt-on alternative. This 28-pin device contains a single IEEE802.3 compatible 10 Base-T MAC/PHY, along with a configurable 8 Kbyte transmit/receive buffer and two programmable register sets for configuring the device (see Figure 14). One of these programmable register sets consists of 4 banks of 32 registers each, and is divided into ETH, MAC, and MII functional groups.

The ETH register group contains the read and write pointers to the 8 Kbyte FIFO, as well as top level control and status registers. The MAC registers are used to configure Media Access Control parameters such as Half/Full Duplex selection, padding, and CRC appending, and the 48-bit MAC address for the device (the MAC address is not configured at the factory, but must be programmed by the user). The MII (Media Independent Interface) registers are used to interface with the second set of programmable PHY registers, i.e., the PHY registers cannot be accessed directly, but are addressed indirectly via the MII address, command and read/write data registers. The 8 Kbyte FIFO is shared by both transmit and receive buffers, and needs to be configured by the user to set the memory ranges for each.

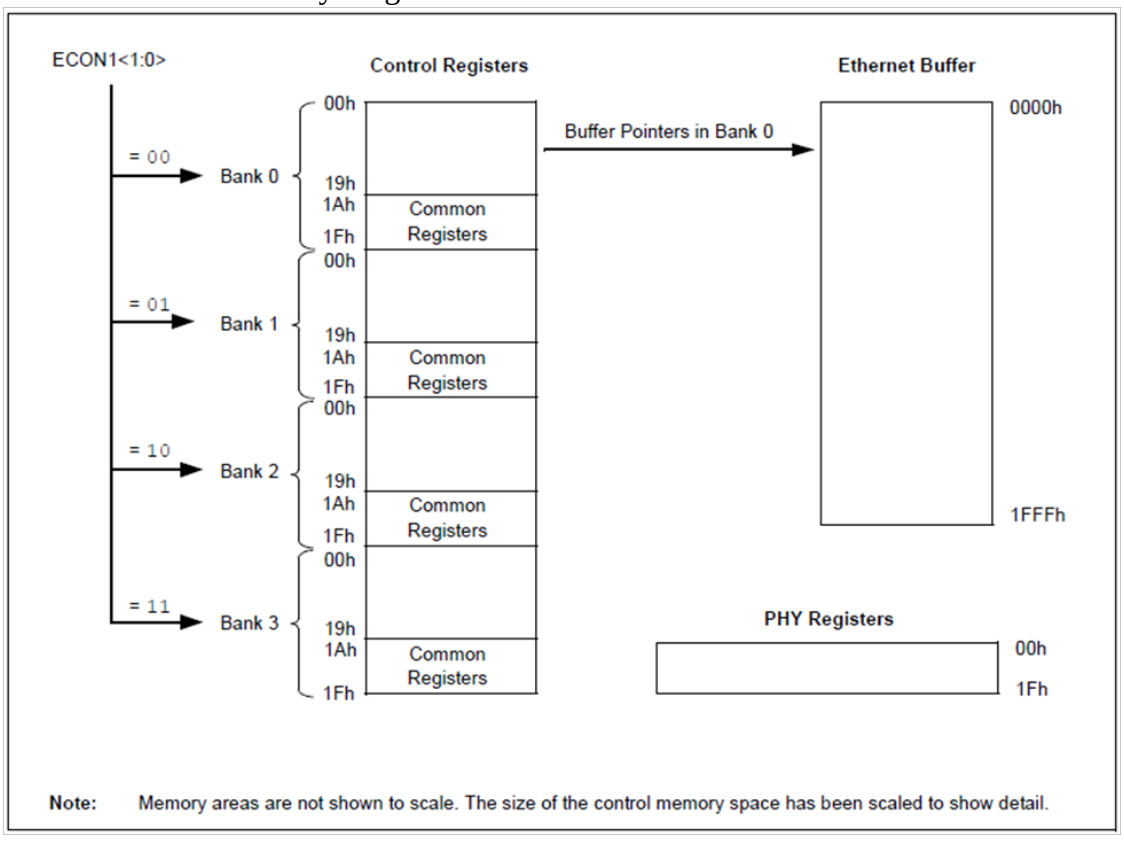


Figure 14: ENC28J60 Control Registers and FIFO [26].

The device requires a 25 MHz input clock, as well as a 3.3 V supply. Interfacing to the ENC28J60 is done via a Serial Peripheral Interface (SPI) bus, which is typically connected to a microcontroller or PIC as the SPI master with the Ethernet controller being the SPI slave. For this project, the DSP will be the SPI master using one of its Serial Control Ports (SCP) configured for SPI communication. The ENC28J60 can operate at a maximum SPI clock speed of 20MHz. The ENC28J60 can generate an interrupt signal for the master device via a dedicated Interrupt output pin, and the source of this interrupt can be six separate on-board conditions, such as Receive Packet Pending and Transmit and Receive Errors. For this project, a complete ENC28J60 circuit board solution manufactured by Olimex was used, which included a crystal oscillator and an RJ-45 connector with magnetics. The circuit board contains stake pin headers for connecting the 3.3 V supply, SPI and Interrupt signals. Figure 15 shows the Ethernet controller jumpered into the DSP evaluation board.

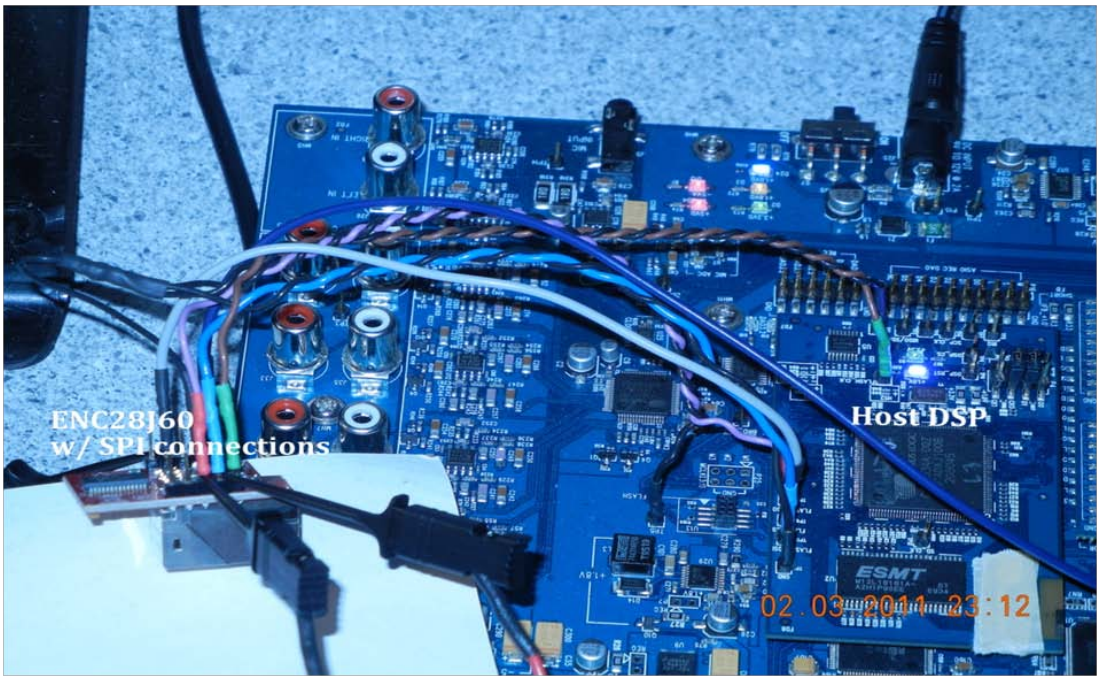

Figure 15: ENC28J60 jumpered into DSP mainboard.

### 4.3 ENC28J60 DEVICE DRIVER

A device driver was written for this project to interface the DSP with the ENC28J60. There existed code previously written for interfacing the DSP with serial flash devices to control the lower level SPI read/write functions, and this code was leveraged for this project. Even though a productizable solution would require the DSP to be a SPI master for both a serial flash (for storing HTML, etc.) as well as the ENC28J60 concurrently, to reduce the complexity of the project, this concurrency requirement was removed. The code to control the reading/writing of the registers contained in the 4 banks of 32 registers, as well as the sequencing of the initialization of the device via programming these registers, was written specifically for this project. Although the device registers can be read before the initialization sequence, the device cannot transmit or receive packets until a couple of dozen specific registers are programmed during the initialization sequence. The register set does not offer a completely generic read/write access API, in that the registers in the ETH register group, the MAC register group, and the MII/PHY register group each have different SPI timing requirements, and so one of the requirements of the device driver was to develop separate handling for each of these 3 register types. The ENC28J60 application note and Errata, as well as example C code from Microchip for initializing the device were all used in the development of the DSP device driver. The Errata in particular requires special attention, in that not all of the features and device initialization steps as detailed in the main app note work correctly, so workarounds are crucial to getting the device properly configured.

The ENC28J60 is controlled via seven different SPI commands [26]:

1) Read Control Register
2) Read Buffer Memory

3) Write Control Register
4) Write Buffer Memory
5) Bit Field Set – Available for ETH registers only (not MAC, MII, PHY or FIFO). Useful for eliminating Read/Modify/Write transactions for some key control registers.
6) Bit Field Clear - Available for ETH registers only (not MAC, MII, PHY or FIFO) Useful for eliminating Read/Modify/Write transactions for some key control registers.
7) System Reset

Even though all registers programmed on the ENC28J60 are 8 bits wide, the Read Control Register command for the MAC and MII registers needs to clock 16 bits for an 8-bit read, with a dummy byte shifted out before the actual data byte.

Application level communication with the DSP is done using the DSP vendor's standard PC console application which utilizes the other Serial Control Port (SCP) on the DSP, that is, one SCP interface is configured for DSP⇔ENC28J60 communication while the other SCP interface is configured for PC⇔DSP communication (see Figure 16). The SCP used for the PC⇔DSP communication is also configured for SPI communication, but on this port, the DSP will be the slave device and the PC will be the master. The DSP device driver written for the ENC28J60 contains an API which can be accessed using the PC console application to aid in the debug and unit testing of the device driver. Specific commands can be sent from the PC to the DSP to initiate ENC28J60 initialization, as well as reading back of the initialized registers for validation.

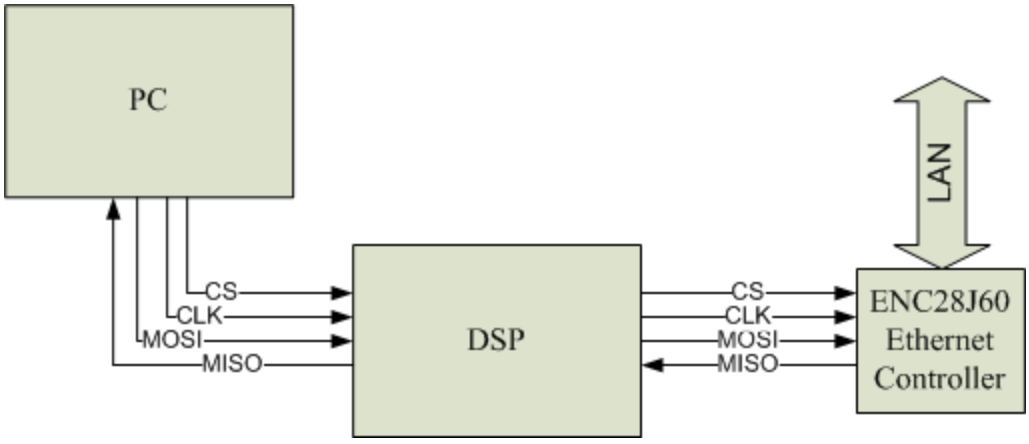


Figure 16: Testbench block diagram showing PC as SPI master to DSP and DSP as SPI master to Ethernet controller.

### 4.4 DEVICE DRIVER BRING-UP AND DEBUGGING

The first step in bring-up of the device driver was to read an Ethernet controller register with a known value so that the communication could be validated. The version code register was chosen which contains a silicon revision number (which in this case was rev B5). A logic analyzer was used to validate the SPI communication and timing parameters. SPI is an industry standard four wire serial bus, with signals for Chip Select, Clock, Master Out Slave In (MOSI), and Master In Slave Out (MISO). Due to the maximum 25 MHz sample rate of the logic analyzer used for debug, the SPI clock was initially set at 8.8 MHz for bring-up. Once the device driver was debugged, the SPI clock was raised to 18.75 MHz to allow for higher data throughput.

Figure 17 shows the 4 lines of the SPI bus during a transaction between the DSP and ENC28J60 captured using a logic analyzer. As mentioned in the previous section, the ETH, MAC, and MII/PHY register each have specific SPI timing requirements, and this was a source of trouble early on in the bring-up of the device driver. The Chip Select (CS) hold time requirement for the MAC register group is 20x that of the ETH registers

group (10 nsec *vs.* 210 nsec), and this finer point from the data sheet was initially overlooked. Read back of the initialized ETH registers yielded a correct result, whereas the read back of the initialized MAC registers was not correct until the proper delay was added to the lower level SPI timing.

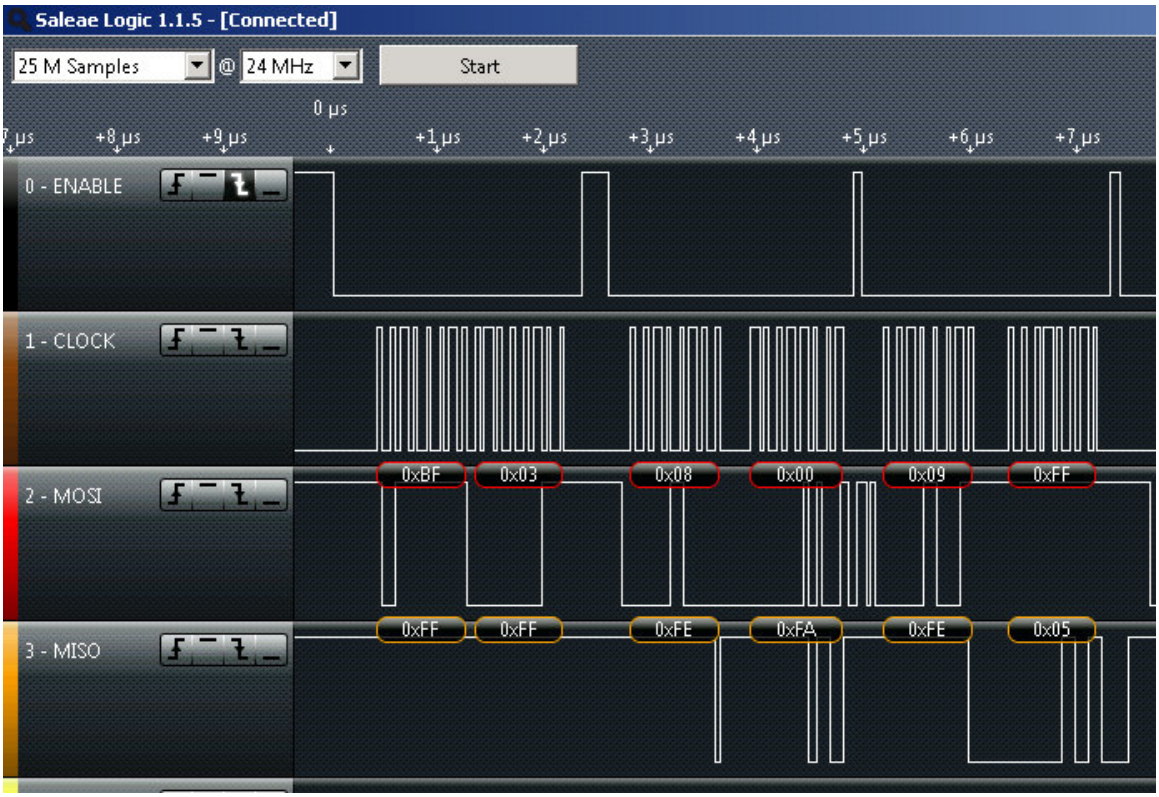


Figure 17: SPI Bus capture of a transaction between the DSP and ENC28J60.

A couple of unit tests were developed which aided in the debugging of the various SPI commands and timings used by the driver. One of the unit tests written performs the initialization sequence, then reads back all of the initialized registers and compares them with the intended results. The other unit test was designed to write the entire 8 Kbyte

FIFO with a random number sequence, then read back the FIFO contents and compare with the known random numbers. This unit test helped debug the bulk data transfer which would be needed for both transmitting and receiving packets.

# Chapter 5: IP Layer

## 5.1 Internet Protocol (IP)

As detailed in the historical overview of Chapter 3, the decision to split out the internetworking IP layer from TCP occurred with v4 [15], several years after the initial development on TCP protocol had begun. This decision was rather crucial in that it allowed for a vastly simplified layer to replace the complicated ARPANET protocols used for message routing. The IP layer is often referred to as the network layer, although what is really meant by this term is a layer which performs "inter-networking", that is, it provides a host-to-host service across heterogeneous networks that allows datagrams to be delivered based on an address without any dependency upon the link layer underneath. The service model is often referred to as "best effort", with no guarantees of delivery. Packets can arrive corrupted, duplicated, and out of order relative to the order they were sent by the source host, as well as simply being lost en route. The following characteristics are often used to describe the IP service model:

- Unreliable – the higher protocol layers are responsible for handling any lost, duplicate, or corrupted packets if reliability is desired.
- Connectionless – no virtual circuit setup, no handshaking between hosts.
- Datagram based – each packet is routed individually.
- Stateless routers – routers (also referred to as gateways) do not contain any connection state, each packet is forwarded based on IP address.
- Link layer agnostic – works on top of any data link layer.

This IP service model is often explained using the postal service as an analogy, with datagrams analogous to the letters which contain an address and dropped into a mail box with no assurance for delivery. Letters are routed amongst postal substations until

reaching the final mail route as datagrams are routed amongst gateways until reaching the final local network, and this routing is based solely on the destination address. The intended applications which are initially envisioned for the DSP will not include any IP routing, and this greatly simplifies the IP layer implementation. Figure 18 shows an IP datagram encapsulated within an Ethernet frame.

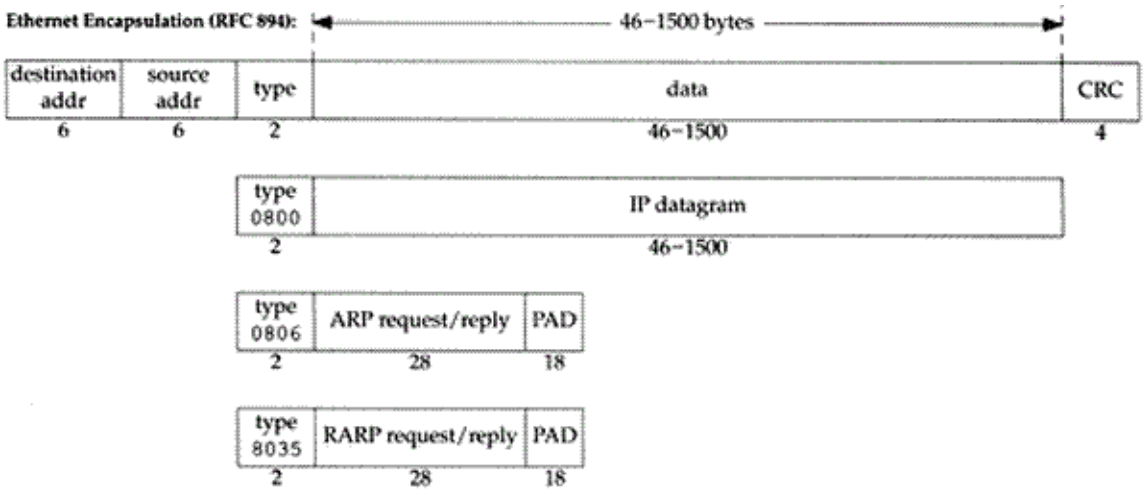


Figure 18: Encapsulation of IP datagram and ARP requests within Ethernet frame [4].

The IP header is shown in Figure 19, and is typically 20 bytes in length if no options are used. The header fields are defined as follows:

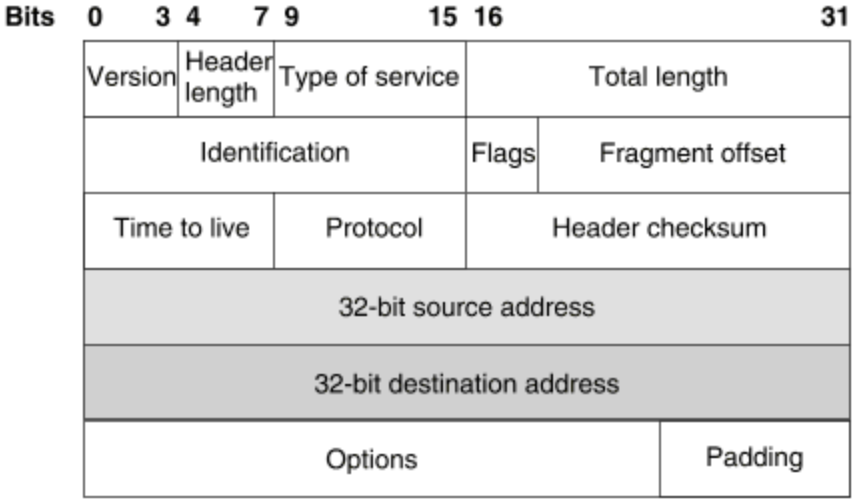


Figure 19: IP datagram header structure [4].

- Version – is the version of IP Protocol used, which is always 0x4 for an IPv4 datagram.
- Header length – this is the length of the IP header in 32-bit words. The minimum value for this is 0x5 (for a 20 byte header with no options). The maximum header length is 60 bytes (40 bytes of options) which will equate to a value of 0xf for the HL field.
- Type of service – this field includes a 3-bit field for specifying precedence, and a 4 bit field for TOS, with bits to minimize delay, maximize throughput, maximize reliability, and minimize monetary cost.
- Total length – this field specifies the combined length of the IP header and the payload.
- Identification – this field contains a unique identifier for each datagram. Typically incremented by 1 for each datagram sent.

- Flags – Two flags and one reserved bit. One flag (DF) to dictate that a datagram is not to be fragmented, and one flag (MF) to denote that there will be more fragments arriving that belong to this datagram.
- Fragment offset – When the MF flag is set, this field specifies the 8-byte chunk offset (e.g. a 512 byte offset will result in a value of 64 for this field) into the total datagram where the data from this fragment belongs.
- Time to live – this field is initialized to a value, typically 64 or 128 which is used as a hop count. Each pass through a router will decrement this field, and if it becomes 0, the datagram will be discarded by the router.
- Protocol – This field denotes the protocol used for the IP datagram payload, and is used as a demux key to determine the higher level protocol which to pass the payload. The protocols we are interested in are UDP (0x11), TCP (0x6), and ICMP (0x1).
- Header checksum – the one's complement of the one's complement sum of the fields in the IP header only (none of the IP datagram payload is included).
- 32-bit source address – the IP address of the host from which this datagram originated.
- 32-bit destination address – the IP address of the host for which this datagram is destined.
- Options – option fields include fields for logging routing addresses and timestamps, etc. These options are rarely used. The options must be padded to a 32-bit length, since the header length is specified in 32-bit words.

## 5.2 IP IMPLEMENTATION

Given the simplicity of the IP layer, particularly since this project did not require any IP routing capability, along with choice to defer the support of fragmentation, the implementation of the IP layer for this project was fairly straightforward and included the following routines:

- IP_Rx_Handler( ) – This routine was called from the Ethernet controller device driver when the decoded EtherType field of the data link layer frame determined that the payload of the frame was an IP datagram (with an ARP packet being the only other supported EtherType). This routine performed the following:
  - Verified that the version field of the IP header was 0x4
  - Performed a range check on the header length field (20<=HL<=60).
  - Performed the checksum on the header itself and compared the result against the arriving checksum. If the checksum did not match, the datagram was dropped.
  - Decoded the protocol field with the result determining which protocol handler (ICMP, UDP, TCP) to pass the IP datagram payload to.
- IP_ Tx_Header_ICMP( ), IP_Tx_Header_UDP( ), IP_Tx_Header_TCP( ) These routines were called by the upper layers to create the IP header to encapsulate an ICMP, UDP, or TCP payload with, by performing the following steps:
  - Populate the source and destination IP address fields.

- Calculate the IP total length field based on the size of the datagram payload and the fixed IP header length (IP options not used).
- Populate the protocol field with the appropriate value: ICMP=0x1, UDP=0x11, TCP=0x6.
- Preincrement and populate the identification field.
- Calculate and populate the IP header checksum field.

- IP_Calc_Dword_Checksum( ) – This routine was the generic routine used by the IP layer as well as by the ICMP, UDP and TCP protocols to calculate the one's complement of the one's complement sum.

## 5.3 Address Resolution Protocol (ARP)

It is important to discuss the other protocol of the TCP/IP suite which can be viewed as being at the same layer as the IP layer, the Address Resolution Protocol (ARP). ARP [30] is the protocol used to resolve the mapping of the 32-bit IP address of a host to the hardware address used by the link layer adapter attached to the host. In our case, with Ethernet used for the link layer, the 32-bit IP address needs to be resolved to a 48-bit hardware (MAC) address used on a local network. As shown in Figure 18, there are two possible protocols of interest carried on top of the link layer, IP and ARP, and these are distinguished by the EtherType field, with 0x800 designating IP and 0x806 designating ARP. The contents of the EtherType field are used as a demux key to determine which protocol at the IP layer will be called to process the payload encapsulated in the Ethernet frame.

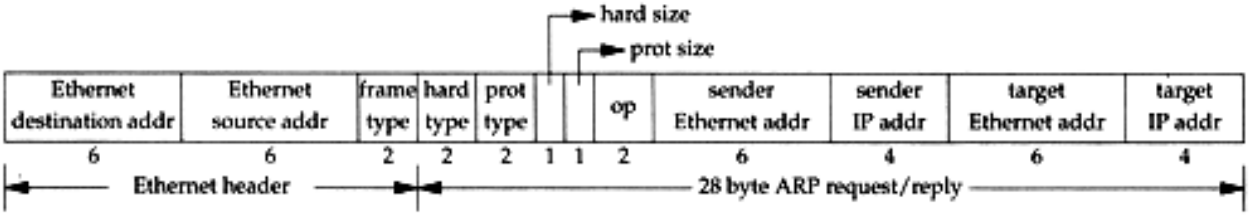


Figure 20: ARP packet format [4]

Figure 20 shows the fields which make up an ARP packet. The field details are as follows:

- Hard type - this 2 byte field uses an enumeration to specify the link layer used, in the case of Ethernet, this value will be 0x0001.
- Protocol type - this 2 byte field uses the EtherType enumeration for the protocol whose addressing needs to be resolved to the hardware layer. Since we are resolving IP addresses, this field will contain the EtherType value for IP, which ix 0x800.
- Hard size - this one byte field specifies the size of the hardware address in bytes, which is 0x6 for Ethernet.
- Protocol size – this one byte field specifies the size of the protocol address in bytes, which is 0x4 for IP.
- Sender Ethernet address – this 6 byte field will contain the MAC address of the hardware interface attached to the host which sent the ARP request/reply packet.
- Sender IP address – this 4 byte field will contain the IP address for the host which sent the ARP request/reply packet.

- Target Ethernet address – this 6 byte field will contain the MAC address of the hardware interface attached to the host for which the ARP request/reply packet is intended.
- Target IP address – this 4 byte field will contain the IP address of the host for which the ARP request/reply packet is intended.

ARP requests utilize the Ethernet broadcast address of ff:ff:ff:ff:ff:ff as the destination MAC address to query all hosts on the local network. The host whose IP address matches the one specified in the query answers the request with an ARP response message specifying its MAC address. ARP requests will be sent as a result of the need to populate the destination MAC address field in the link layer header for any outgoing packet, if the MAC address is not already known by the sending host. Hosts will typically maintain an ARP cache, placing IP_address:MAC_address tuples in the cache when it receives an ARP response, alleviating the need for an ARP request for every packet transmitted. Figure 21 shows a Wireshark capture of an ARP request/reply sequence between the DSP and a PC on its local network. Due to the simple structure of the ARP request packet, this was the first unit test developed for exercising the packet transmit code on the DSP.


Figure 21: ARP request packet and response between DSP and local PC.

### 5.4 INTERNET CONTROL MESSAGE PROTOCOL (ICMP)

The Internet Control Message Protocol (ICMP) is a protocol for sending error and routing diagnostic messages between hosts [29]. Although this protocol is encapsulated

in an IP datagram, it is not typically viewed as belonging to the transport layer (Figure 9 shows ICMP as being at the top end of the IP layer). ICMP messages are grouped into two distinct classes [21]:

- ICMP error messages:
  - Destination Unreachable
  - Redirect
  - Source Quench
  - Time Exceeded
  - Parameter Problem
- ICMP query messages:
  - Echo
  - Information
  - Timestamp
  - Address Mask

The most common ICMP application users are familiar with is the ping program, which uses the echo query message. Figure 22 shows the ICMP message format for the echo message. The identifier field is commonly used to hold the process ID of the sending host, and this is required to be echoed back, so that it can be checked in the echo response. The sequence number is a value which will normally increase by one for each message sent.

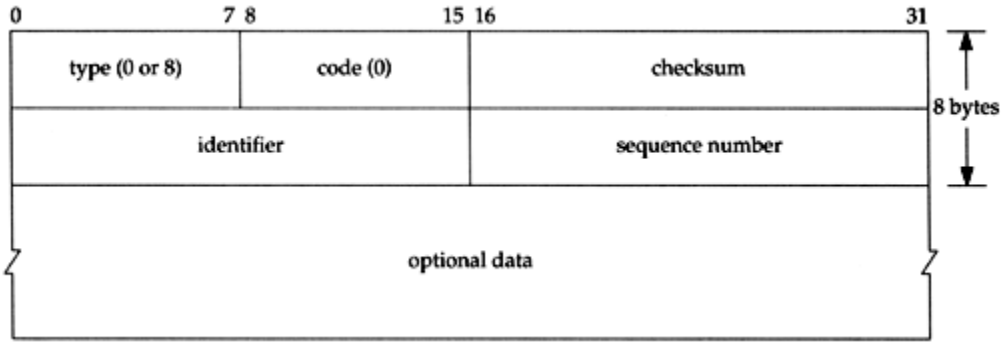

Figure 22: ICMP query message structure [4].

Once the unit test for the ARP request/reply was completed, the next easiest unit test to develop was to issue an ICMP echo message to a PC on the local network. Figure 23 shows the Wireshark capture of an echo query issued from the DSP, which in turn triggers an ARP request from the PC to the DSP. The DSP answers the request with an ARP response. On receiving the ARP response, the PC then knows the MAC address needed to reply to the echo query message, and responds with the echo reply.

| | | | | | |
|---|---|---|---|---|---|
| 5 | 10.353100 | 10.15.59.125 | 10.15.59.173 | ICMP | Echo (ping) request |
| 6 | 10.353155 | Wistron_15:79:06 | Broadcast | ARP | Who has 10.15.59.125? Tell 10.15.59.173 |
| 7 | 11.810042 | Crystals_ab:ea:cc | Wistron_15:79:06 | ARP | 10.15.59.125 is at 00:60:11:ab:ea:cc |
| 8 | 11.810064 | 10.15.59.173 | 10.15.59.125 | ICMP | Echo (ping) reply |

Figure 23: ICMP echo query message triggering ARP request/reply sequence

# Chapter 6: Transport Layer

## 6.1 User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) provides a simple, connectionless, unreliable protocol designed to run on top of the IP layer for datagram based communication between host processes [31]. Figure 24 shows a UDP datagram encapsulated within an IP datagram.

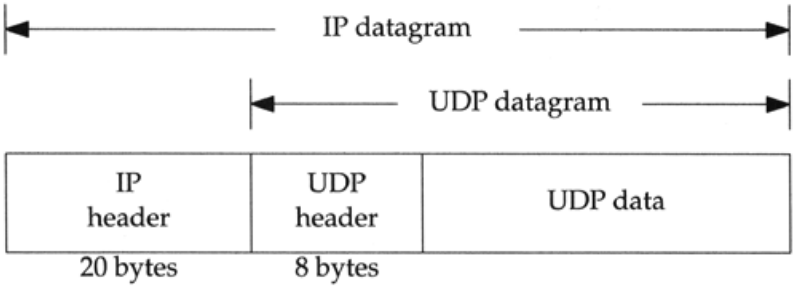


Figure 24: UDP datagram encapsulated within IP datagram [4].

Hosts can support multiple communicating processes using a single IP address, through the use of 16-bit port number to identify a process executing on a host. Continuing the postal service analogy from Chapter 5 that was used to explain the IP service model, the addition of a port at the transport layer is analogous to how a box number is used to differentiate amongst multiple potential recipients at the same address. The IP_address:Port_number tuple is referred to as a socket, and the pair of sockets uniquely identifies the communicating processes. Ports are divided into two classes, the "well known" port numbers associated with specific server applications such as an echo server or bootstrap protocol, and "ephemeral" ports which are used by the client application to distinguish amongst multiple possible connections. Well-known port

numbers range from 0-124, and the Internet Assigned Numbers Authority specifies that: “The Dynamic and/or Private Ports are those from 49152 through 65535” [32].

A client is required to specify the IP address and port number for the server with which it wishes to communicate. A server does not need to specify the host address or port number of clients from which it will accept datagrams, but only needs to specify the port number on which it will listen. Note that ‘sockets’ is an overloaded term as used in networking literature. As just detailed, it is commonly used to describe the IP_address:Port_number tuple, but it is also used to refer to the “layer” which is used by applications to interface with the transport layer. This “socket layer” includes functions in the TCP/IP code stack which are called to setup a socket (IP_address:Port_number tuple), while specifying whether the stream type is datagram or byte stream, as well as functions to connect to a host (for client processes), and bind to a port and listen for arriving datagrams (for server processes).

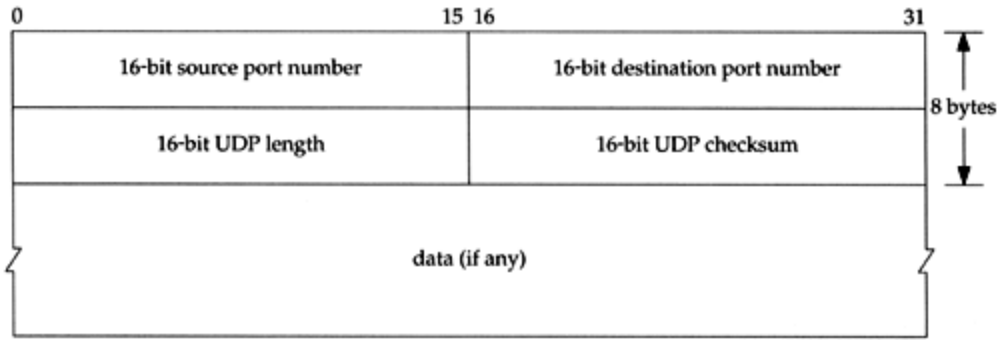


Figure 25: UDP header details [4].

Figure 25 shows the fields of the fixed length UDP header. The header fields are defined as follows:

- Source port – This will be populated with the well-known port number for datagrams coming from the server process, and with an ephemeral port number for datagrams coming from the client process.
- Destination port – This will be populated with the ephemeral port of the destination client process for datagrams coming from the server process, and with the well-known port number on which the server process is listening for datagrams coming from the client.
- UDP length – This field contains the length in bytes of the entire UDP datagram, and spans the 8 byte header plus the data payload. Note that this field is somewhat redundant in that the UDP length can be inferred from the IP total length along with the fixed length UDP header. The length field, since it is included in the checksum, provides an added robustness against corrupt packets.
- UDP checksum – This field contains the IP checksum (one's complement of the one's complement sum) of not only the UDP header plus data payload, but also includes a prepended pseudo-header with fields from the IP header. Figure 26 shows the fields used for the checksum calculation. The use of the checksum is optional with UDP, but strongly recommended. If the checksum is not used, the field is populated with zero. If the resulting checksum is zero, the value of 0xffff is used (this leverages the one's complement arithmetic property of having two representations for zero). Note that the checksum field is populated with zero **before** the checksum is calculated. Also note that the pseudo-header is not actually transmitted, but only used for the checksum calculation.

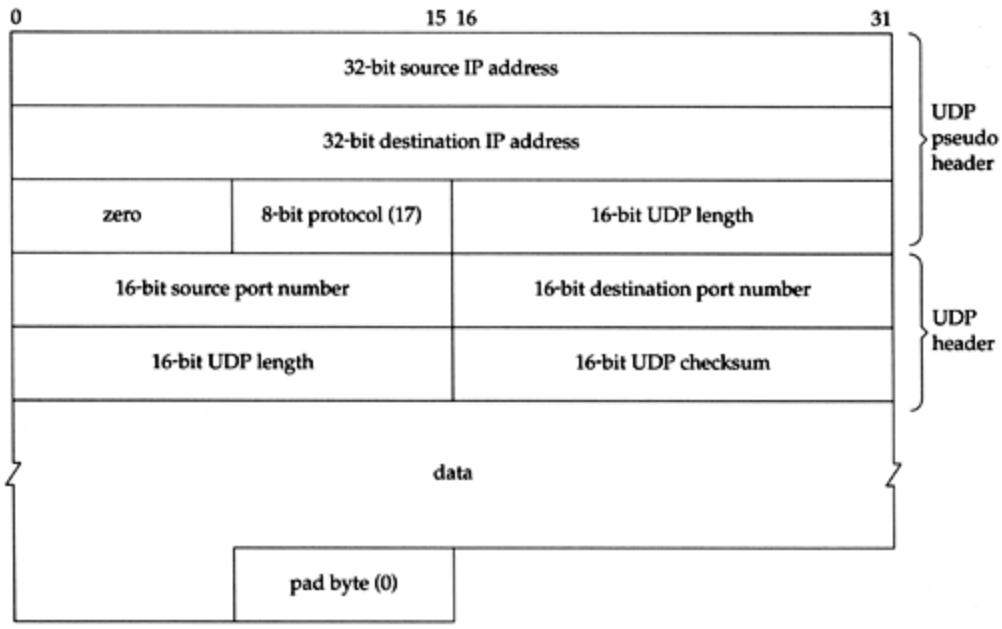


Figure 26: Fields used in UDP checksum calculation [4].

## 6.2 UDP UNIT TEST

A simple UDP echo server unit test was developed to validate the routines implemented to support UDP. This involved the following:

- Listen on port 7 (echo port) – datagrams passed from IP layer will have checksum validated, and then check to see if the destination port is 0x7.
- If destination port is 0x7, form the echo response by swapping source/destination IP addresses, ports and MAC addresses of the received packet and transmit back out.

Figure 27 shows the Wireshark capture of the UDP echo server traffic between the DSP and PC on the local network. The use of the software utility EchoTool [33] was used to generate the echo request packets on the PC.

| No. | Time | Source | Destination | Protocol | Info |
|---|---|---|---|---|---|
| 1 | 0.000000 | 10.15.59.173 | 10.15.59.125 | ECHO | Request |
| 2 | 0.000325 | 10.15.59.125 | 10.15.59.173 | ECHO | Response |
| 3 | 0.101753 | 10.15.59.173 | 10.15.59.125 | ECHO | Request |
| 4 | 0.102272 | 10.15.59.125 | 10.15.59.173 | ECHO | Response |
| 5 | 0.204279 | 10.15.59.173 | 10.15.59.125 | ECHO | Request |
| 6 | 0.204616 | 10.15.59.125 | 10.15.59.173 | ECHO | Response |
| 7 | 0.306598 | 10.15.59.173 | 10.15.59.125 | ECHO | Request |
| 8 | 0.306927 | 10.15.59.125 | 10.15.59.173 | ECHO | Response |
| 9 | 0.408153 | 10.15.59.173 | 10.15.59.125 | ECHO | Request |
| 10 | 0.408467 | 10.15.59.125 | 10.15.59.173 | ECHO | Response |
| 11 | 0.509705 | 10.15.59.173 | 10.15.59.125 | ECHO | Request |
| 12 | 0.510051 | 10.15.59.125 | 10.15.59.173 | ECHO | Response |

Figure 27: Wireshark capture of UDP Echo server transactions.

### 6.3 Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) is the cornerstone protocol enabling reliable communication between host processes, even though it runs on top of the unreliable IP layer. Figure 28 shows a TCP segment encapsulated within an IP datagram.

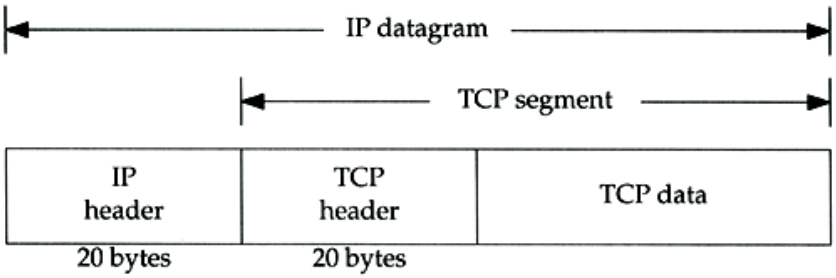


Figure 28: TCP segment encapsulated within IP datagram [4].

This is essentially identical to the UDP encapsulation shown in Figure 24, but with the larger TCP header length (minimum of 20 bytes for TCP *vs.* fixed 8 bytes for UDP). These extra fields in the TCP header are the key elements enabling the reliable communication. Figure 29 shows details the TCP header fields.

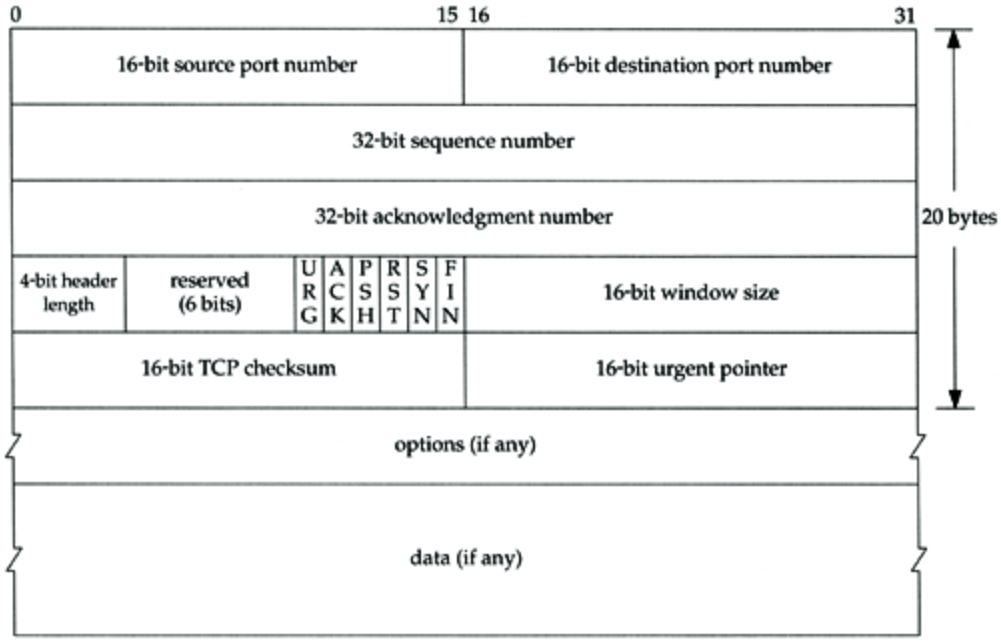


Figure 29: TCP header details [4].

The fields of the TCP header are defined as follows:

- Source port – This is identical to the source port definition for UDP as detailed in Section 6.1.
- Destination port – This is identical to the destination port definition for UDP as detailed in Section 6.1.
- Sequence number – The 32-bit sequence number is used to track each byte of a byte stream. The sequence number of a particular sequence denotes the **starting** byte of the data payload.
- Acknowledgment number – The 32-bit acknowledgement number is used denote the **next** byte that is expected to be received.

- Header length – This is also referred to as the data offset field, and it denotes the length of the header in 32-bit words. Valid values range from the minimum of 0x5 (20 bytes) to maximum of 0x3c (60 bytes).
- Flags – the URG, ACK, PSH, RST, SYN, FIN flags are used to define the state of the TCP connection. These will be detailed below.
- Window size – The window size field is used to communicate the amount of data that the receiver is able to accept. Hence, the windows size is used for flow control. The receiver will modulate this value and communicate it back to the sender as its internal buffers fill up when receiving segments, and as they are emptied by the upstream application consuming the data.
- TCP checksum – This field is similar to the checksum calculated for UDP. It also uses a pseudo-header pre-pended for the checksum calculation. Unlike UDP checksums, however, TCP checksums are mandatory.
- Urgent pointer – This field is used in combination with the URG flag to denote an offset into the data payload of the **ending** byte of data within the segment which should be passed to the application immediately.
- Options – Options exist for timestamps, selective acknowledgements, window size scale factors, and Maximum Segment Size (MSS). MSS is the only option we are currently interested in, and is used to notify the sender the largest segment the receiver is willing to accept.

The state machine for TCP is rather complex, with 11 states. These states are defined in RFC 793 as follows [2]:

LISTEN - represents waiting for a connection request from any remote TCP and port.
SYN-SENT - represents waiting for a matching connection request after having sent a connection request.
SYN-RECEIVED - represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.
ESTABLISHED - represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.
FIN-WAIT-1 - represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.
FIN-WAIT-2 - represents waiting for a connection termination request from the remote TCP.
CLOSE-WAIT - represents waiting for a connection termination request from the local user.
CLOSING - represents waiting for a connection termination request acknowledgment from the remote TCP.
LAST-ACK - represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (which includes an acknowledgment of its connection termination request).
TIME-WAIT - represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.
CLOSED - represents no connection state at all.

The state of a TCP connection will transition among these states as shown in Figure 30.

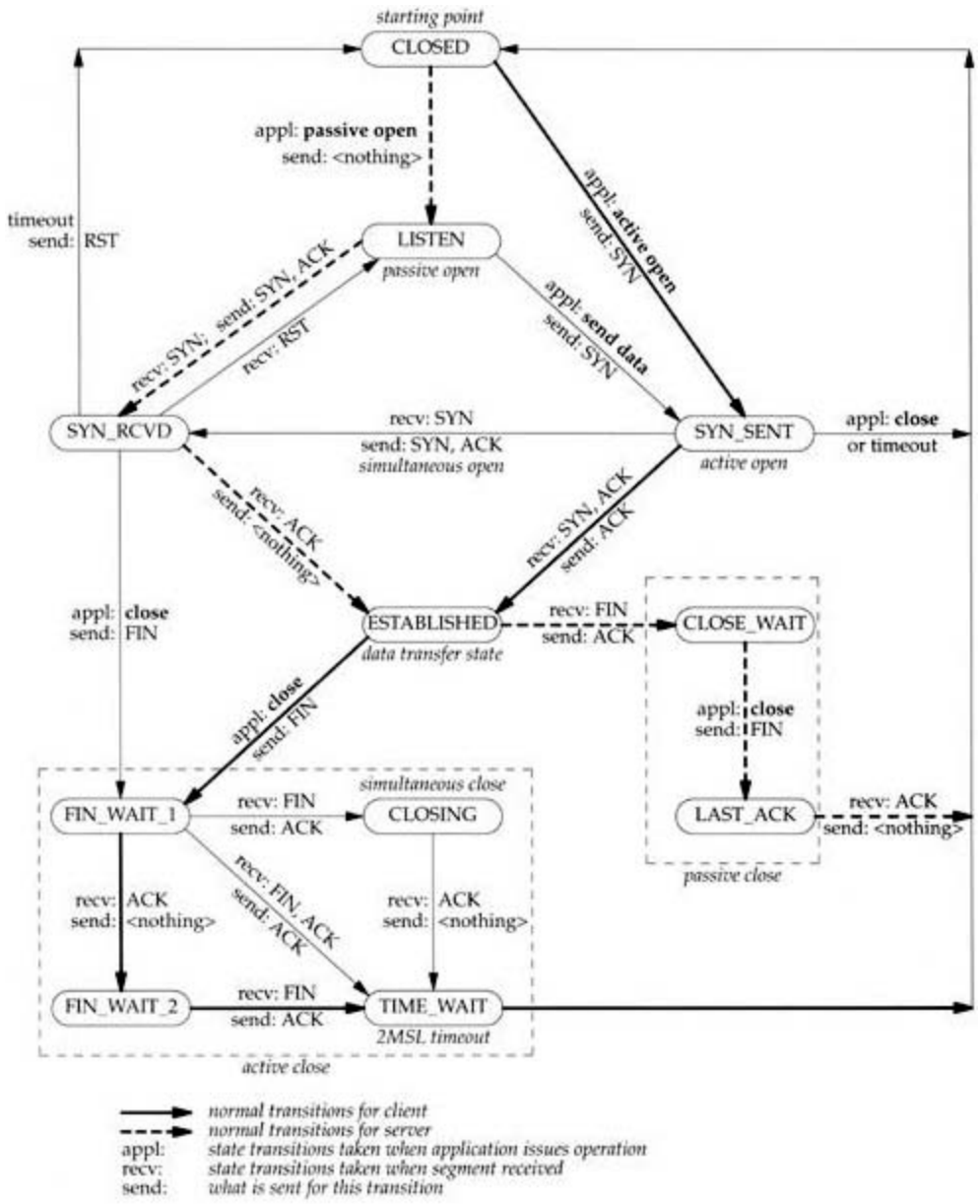


Figure 30: TCP state transition diagram [34].

TCP is a connection oriented protocol, and uses a 3-way handshake to establish the connection. The client process will perform what is referred to as the “active open”,

initiating the connection by sending a "SYN" (synchronize) segment. The SYN segment will have the SYN bit set, the port number for the server process and the ephemeral port number for the client-side process specified, the Window size specified, the Sequence Number initialized, and possibly the Maximum Segment Size specified in the options field. Note that even though there is no data in the SYN segment itself, the SYN segment consumes one byte of the sequence space. The server process performs a passive open by listening for arriving SYN segments, responding with a SYN/ACK segment, i.e., both SYN and ACK bits will be set, and will use the Sequence Number from the client + 1 as its Acknowledgment number and send its initial Sequence Number back to the client (along with Window size, etc.). The final step is for the client to respond back to the server with an ACK segment which will contain the server's initial Sequence Number + 1 as its Acknowledgment number. At this point the connection is considered to be in the established state. The top of Figure 31 shows the timeline of the 3-way handshake, with example Sequence and Acknowledgement numbers.

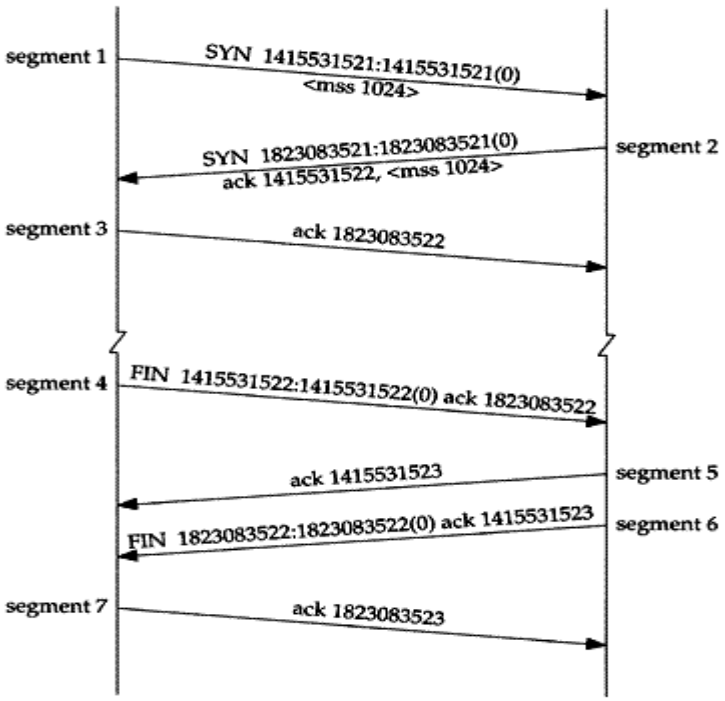


Figure 31: TCP connection establishment and termination [4].

Once a connection is established data can flow in both directions of the connection, i.e., it is a full duplex connection. Unlike the datagram oriented UDP protocol, TCP uses the established connection to allow a byte stream to flow between the host processes. Since the connection is full duplex, each side of the connection must be terminated separately with a 2-way handshake using FIN (finish) and ACK segments. Like the SYN segments, FIN segments also consume one byte of the sequence space. The lower part of Figure 31 shows the connection termination sequence.

Reliability is achieved through the use of the sequence and acknowledgement numbers for each direction of the connection. If an acknowledgment for a segment is not received within the specified time-out period, then the segment is re-transmitted. In its

original specification, TCP only supported a cumulative acknowledgement, i.e. it did not provide for any negative or selective acknowledgments. There was no provision for using a negative acknowledgment for the receipt of a corrupted segment, or a selective acknowledgment for indicating that all but the first segment of a sliding window had been received.

Flow control is achieved by each end of the connection advertising a window size, which specifies the number of bytes that each host is currently willing to accept. This window size is transmitted back to the transmitter along with every ACK. A TCP implementation will maintain a Protocol Control Block for each connection, with variables to keep track of the connection state, sequence and acknowledgment numbers for each side of the connection, etc. Figure 32 shows the send side state for an example connection.

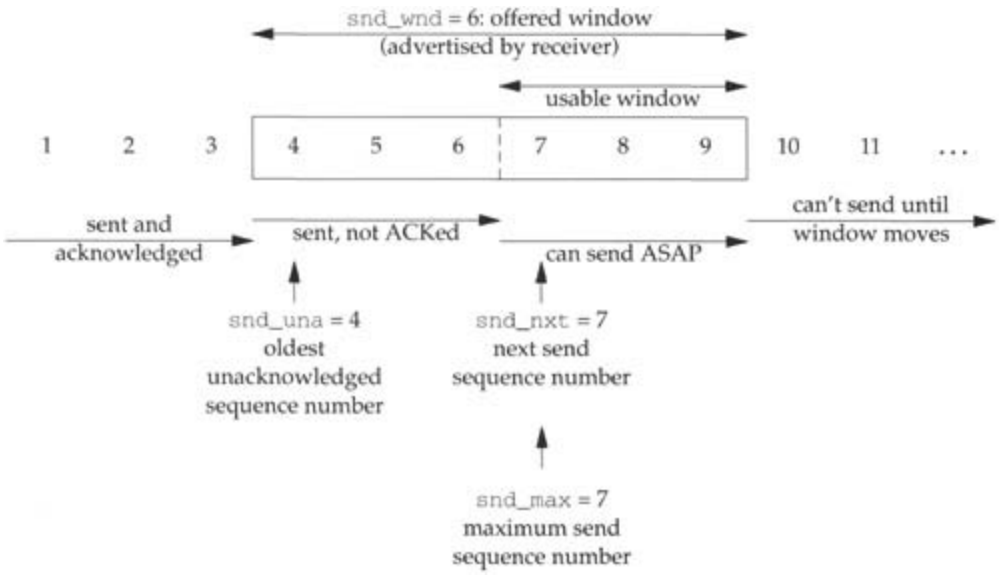


Figure 32: TCP send sequence variables [34].

Using the variables specified, a valid ACK segment for the current state of a connection would require:

snd_una < Acknowledgement number <= snd_max

i.e., if the received Acknowledgement number was <= snd_una, then it would be a duplicate ACK, and if the received Acknowledgement number > snd_max, then that would indicate an ACK for a segment not yet sent.

Also, conditions where snd_nxt < snd_max will indicate that a re-transmission is occurring [34].

Figure 33 shows the receive side state for an example connection.

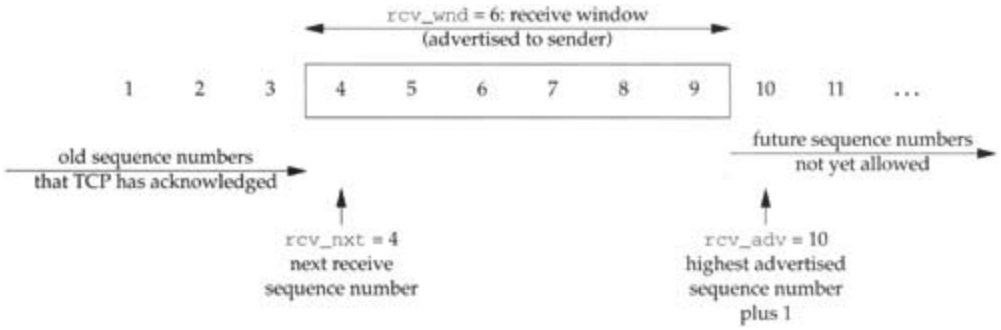


Figure 33: TCP receive sequence variables [34].

For the receive side, valid values for rcv_nxt will be:

rcv_nxt <= Sequence number < rcv_nxt + rcv_wnd

i.e., if the received Sequence number < rcv_nxt, then that would indicate a duplicate segment was received, and if Sequence number => rcv_nxt + rcv_wnd, that would indicate the receipt of a segment outside of the advertised window.

## 6.4 TCP IMPLEMENTATION

Since the TCP protocol entails a much more complex state machine and dynamic memory management, and given that complete implementations of TCP alone are on the order of 4500 lines of C code [34], only a small portion of the needed TCP routines were designed and developed for the DSP. Some details of the status of the design and implementation are given below.

As mentioned in Section 6.3, the suggested design from RFC 793 of using a Protocol Control Block as the key structure for keeping state information for a TCP connection will also be used in the completion of this project [2].

> The maintenance of a TCP connection requires the remembering of several variables. We conceive of these variables being stored in a connection record called a Transmission Control Block or TCB. Among the variables stored in the TCB are the local and remote socket numbers, the security and precedence of the connection, pointers to the user's send and receive buffers, pointers to the retransmit queue and to the current segment. In addition several variables relating to the send and receive sequence numbers are stored in the TCB.
>
> Send Sequence Variables
>
> SND.UNA - send unacknowledged
> SND.NXT - send next
> SND.WND - send window
> SND.UP - send urgent pointer
> SND.WL1 - segment sequence number used for last window update
> SND.WL2 - segment acknowledgment number used for last window update
> ISS - initial send sequence number
>
> Receive Sequence Variables
>
> RCV.NXT - receive next
> RCV.WND - receive window
> RCV.UP - receive urgent pointer
> IRS - initial receive sequence number

The BSD and lwIP implementations include quite a few other variables in their PCBs than those just quoted from RFC 793. For instance, there are additional variables that cover the state of the timer related activities for the retransmission of segments. The PCB structure which will be used in the completion of this project is still being designed.

The dynamic memory allocation scheme for segments which will be used in the completion of the project will be of the "bucket" type which makes use of a pool of fixed size buffers which is much simpler and can yield a more deterministic mallocation time than a generic malloc routine.

The following routines for handling TCP were partially completed:

- X_S_TCP_Rx_Handler( ) – This routine handles the initial processing for a received segment. The first step is to read in the minimum 5 dword TCP header length from the Ethernet controller FIFO. It will then determine the length of any TCP header options which need to be read by examining the header length (or data offset) field in the header. Since the length of the data payload is not specified as it is with UDP, the header length field will then also be used in conjunction with the total length field of the IP header to determine the length of the data payload of the TCP segment to be read in. Then the pseudo-header, which includes fields from the IP header, is created and pre-pended to the TCP header, and the checksum is performed for the segment. The segment is dropped if the checksum fails. Code still needing development is the copying of the pertinent header fields into the PCB for this connection, as well as the code to find the PCB for the connection in a list if there is to be support for more than one connection.

- X_S_TCP_Tx_Handler( ) – This routine is used to construct the TCP header for a segment to be transmitted. A pointer to a PCB is passed in which will then be used to populate the fields for the source and destination ports, the segment and acknowledgement numbers, and the flags and window size. The pseudo-header is then constructed, the checksum calculated and used to populate the checksum field in the header.
- X_S_TCP_Generate_Ephemeral( ) – This routine will generate an ephemeral port number for use by client processes. A random number is masked with 0x3fff and added to 0xc000, to create a port number in the range of 49152-65535 as specified by IANA [32].

# Chapter 7: Conclusion and Future Work

This project and report have demonstrated that it is possible, although involved, to develop a TCP/IP stack from the ground up for a DSP using the native assembly language of the device. The choice to write the protocol stack as opposed to porting an existing one such as lwIP was made given the constraints of an unproven and inefficient C compiler, the expectations such a generic library has regarding interfacing with an OS API, and most importantly, the need to tailor the implementation to be as optimized as possible for a given application. The rewards of having network connectivity for such a device, without the need for a host microcontroller, however, are potentially far reaching in terms of the application space that such a device could address. Although only a fraction of a complete TCP/IP protocol stack was implemented for this project, the project demonstrated that such an approach can result in a productizable solution.

The status of the implementation of the TCP/IP stack is detailed below. Link layer and IP layer implementations are fairly complete with the exception of fragmentation and re-assembly. The design choice to not support routing simplified the IP layer implementation.

- Ethernet Controller device driver and link layer – 90% complete. Robustness handling for Ethernet controller error conditions is yet to be added.
- IP layer
    - IP datagram transmit and receive - 75% complete. Support for fragmentation and re-assembly needed.
    - ARP cache needs to be expanded beyond single entry, and time-outs are needed for cache entries.

- Transport layer
    - UDP – need to incorporate Protocol Control Buffers (PCBs) to enable more than a single connection.
        - DHCP implementation 75% complete
    - TCP – transmit and receive header handling 50% complete

Below is an estimate of the development time needed to achieve TCP connection establishment/termination, sliding window, segment reordering and re-transmission.

- PCB – Protocol Control Buffers – 4 days
- Bucket memory allocation scheme – 5 days
- Timers – 3 days
- Initial Sequence Number generation – 3 days
- TCP connection and termination – 4 days
- Sliding window – 5 days
- Segment reordering – 3 days
- Re-transmission – 3 days

# Bibliography


[1] Postel, J. 1981. RFC 791 Internet Protocol. http://tools.ietf.org/html/rfc791

[2] Postel, J. 1981. RFC 793 Transmission Control Protocol. http://tools.ietf.org/html/rfc793#page-15

[3] Postel, J. 1979. IEN 98 TCP Implementation Status. http://www.postel.org/ien/txt/ien98.txt

[4] Stevens, W. 1993. TCP/IP Illustrated, Volume 1: The Protocols, Addison-Wesley Professional.

[5] Announcement of iPic small TCP/IP implementation. 1999. http://www.ietf.org/mail-archive/web/ietf/current/msg10705.html

[6] iPic. 1999. http://www.sciencedaily.com/releases/1999/08/990811075627.htm

[7] Texas Instruments press release. 2001. Network Developers Kit for TMS3206000. http://www.ti.com/sc/docs/news/2001/01090.htm

[8] Texas Instruments Luminary Micro 100 c ycle flash issue. 2011. http://e2e.ti.com/support/microcontrollers/stellaris_arm_cortex-m3_microcontroller/f/471/t/95905.aspx

[9] Texas Instruments Network Developers Kit Overview. www.dspvillage.com/pdfs/ndk_tcpip_ov.pdf

[10] Dunkels, A. 2001. Design and Implementation of the lwIP TCP/IP Stack. Swedish Institute of Computer Science.

[11] Dunkels, A. 2003. Full TCP/IP for 8-bit architectures. Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MobiSys `03).

[12] Scheblik, T. 1968. RFQ No. DAHC15 69 Q 0002. http://www.cs.utexas.edu/users/chris/DIGITAL_ARCHIVE/ARPANET/RFQ-ARPA-IMP.pdf

[13] Heart, F. 1978. ARPANET Completion Report, Report No. 4799.

[14] Bolt Beranek and Newman, Inc. 1973. Interface Message Processor -- Specifications for the Interconnection of a Host and an IMP. BBN Report No. 1822.

[15] Leiner B. A Brief History of the Internet. http://www.isoc.org/internet/history/brief.shtml#Origins

[16] McKenzie, A. 1972. HOST/HOST Protocol for the ARPA Network. Current Network Protocols, Network Information Center, Stanford Research Institute, Menlo Park, California, (NIC8246).

[17] Bolt Beranek and Newman, Inc. 1974. Network Design Issues Report. BBN Report No. 2918.

[18] Cerf, V. 1974. RFC 635 An Assessment of ARPANET Protocols. http://tools.ietf.org/html/rfc635

[19] Cerf, V., Kahn, R. 1974. A Protocol for Packet Network Intercommunication, IEEE Trans on Comms, Vol Com-22, No 5.

[20] Postel, J. 1981. RFC 801 NCP/TCP Transition Plan. http://www.ietf.org/rfc/rfc801.txt

[21] Brandon, R. 1989. RFC 1122 Requirements for Internet Hosts -- Communication Layers. http://tools.ietf.org/html/rfc1122

[22] International Organization for Standardization. 1984. ISO 7498:1984 Information processing systems -- Open Systems Interconnection -- Basic Reference Model.

[23] Tanenbaum, A. 2002. Computer Networks, 4th ed. Prentice Hall.

[24] Microsoft Corporation. TCP/IP Protocol Architecture. http://technet.microsoft.com/en-us/library/cc958821.aspx

[25] Simmons, M. 2008. AN1120 Ethernet Theory of Operation. Microchip Technology, Inc. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en533903

[26] Microchip Technology, Inc. 2008. ENC28J60 Data Sheet Stand-Alone Ethernet Controller with SPI Interface. http://ww1.microchip.com/downloads/en/DeviceDoc/39662c.pdf

[27] Microchip Technology, Inc. 2006. ENC28J60 Rev. B5 Silicon Errata http://ww1.microchip.com/downloads/en/DeviceDoc/80264d.pdf

[28] Digital Equipment Corporation, Intel Corporation and Xerox Corporation. 1982. The Ethernet, A Local Area Network. Data Link Layer and Physical Layer Specifications. http://decnet.ipv7.net/docs/dundas/aa-k759b-tk.pdf

[29] Postel, J. 1981. RFC 792 Internet Control Message Protocol. http://tools.ietf.org/html/rfc792

[30] Plummer, D. 1982. RFC 826 An Ethernet Address Resolution Protocol. http://tools.ietf.org/html/rfc826

[31] Postel, J. 1980. RFC 768 User Datagram Protocol. http://tools.ietf.org/html/rfc768

[32] Internet Assigned Numbers Authority - Assigned Port numbers. http://www.iana.org/assignments/port-numbers

[33] EchoTool - Echo client and server. http://bansky.net/echotool/

[34] Stevens, W. 1995. TCP/IP Illustrated, Volume 2: The Implementation. Addison-Wesley Professional.